UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA E INFORMÁTICA INDUSTRIAL

ADRIANO FRANCISCO RONSZCKA

MÉTODO PARA A CRIAÇÃO DE LINGUAGENS DE PROGRAMAÇÃO E COMPILADORES PARA O PARADIGMA ORIENTADO A NOTIFICAÇÕES EM PLATAFORMAS DISTINTAS

TESE DE DOUTORADO

CURITIBA
2019

ADRIANO FRANCISCO RONSZCKA

MÉTODO PARA A CRIAÇÃO DE LINGUAGENS DE PROGRAMAÇÃO E COMPILADORES PARA O PARADIGMA ORIENTADO A NOTIFICAÇÕES EM PLATAFORMAS DISTINTAS

Tese de doutorado apresentada ao Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial da Universidade Tecnológica Federal do Paraná como requisito parcial para obtenção do título de “Doutor em Ciências” – Área de Concentração: Engenharia da Computação.

Orientador: Prof. Dr. Jean Marcelo Simão
Co-orientador: Prof. Dr. João Alberto Fabro

CURITIBA
2019

Dados Internacionais de Catalogação na Publicação

Ronszcka, Adriano Francisco
Método para a criação de linguagens de programação e compiladores para o paradigma orientado a notificações em plataformas distintas [recurso eletrônico] / Adriano Francisco Ronszcka.-- 2019.
1 arquivo texto (375 f.) : PDF ; 11,4 MB.

Modo de acesso: World Wide Web.
Texto em português com resumo em inglês.
Tese (Doutorado) - Universidade Tecnológica Federal do Paraná. Programa de Pós-graduação em Engenharia Elétrica e Informática Industrial. Área de Concentração: Engenharia de Computação, Curitiba, 2019.
Bibliografia: f. 326-339.

1. Engenharia elétrica - Teses. 2. Paradigma orientado a notificações. 3. Linguagem de programação (Computadores). 4. Compiladores (Computadores). 5. Software - Desenvolvimento. 6. Processamento paralelo (Computadores). 7. Plataforma (Computação). 8. Simulação (Computadores). 9. Engenharia de computador. I. Simão, Jean Marcelo, orient. II. Fabro, João Alberto, coorient. III. Universidade Tecnológica Federal do Paraná. Programa de Pós-graduação em Engenharia Elétrica e Informática Industrial. IV. Título.

CDD: Ed. 23 -- 621.3

Biblioteca Central do Câmpus Curitiba - UTFPR
Bibliotecária: Luiza Aquemi Matsumoto CRB-9/794

UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

CAMPUS CURITIBA

TERMO DE APROVAÇÃO DE TESE Nº 194

A Tese de Doutorado intitulada **"Método para a Criação de Linguagens de Programação e Compiladores para o Paradigma Orientado a Notificações em Plataformas Distintas",** defendida em sessão pública pelo(a) candidato(a) **Adriano Francisco Ronszcka**, no dia **24 de maio de 2019**, foi julgada para a obtenção do título de Doutor em Ciências, **área de concentração Engenharia de Computação**, e aprovada em sua forma final, pelo Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial.

BANCA EXAMINADORA:

Prof(a). Dr(a). Paulo Cézar Stadzisz - Presidente – (UTFPR)
Prof(a). Dr(a). Fabrício Enenbreck – (PUC-PR)
Prof(a). Dr(a). Murilo Vicente Gonçalves da Silva – (UFPR)
Prof(a). Dr(a). André Murbach Maidl – (Elastic)
Prof(a). Dr(a). André Schneider de Oliveira - (UTFPR)
Prof(a). Dr(a). João Alberto Fabro – (UTFPR)

A via original deste documento encontra-se arquivada na Secretaria do Programa, contendo a assinatura da Coordenação após a entrega da versão corrigida do trabalho.

Curitiba, 24 de maio de 2019.

AGRADECIMENTOS

Primeiramente, o autor deste documento de tese apresenta seus agradecimentos de ordem pessoal. Tais agradecimentos naturalmente vão primeiramente aos pais, Evilasio Ronszcka e Maria Doracilda Ruthes Ronszcka, que são a origem e o presente de tudo, bem como aos avôs e avós, assim como aos ancestrais que os precederam, sendo eles a origem da origem. Outrossim, certamente que os agradecimentos também alcançam a companheira de jornada, Jociene de Andrade Feitosa, incluindo sua inestimável paciência e afetividade. Ainda, os agradecimentos se estendem absolutamente aos amigos de jornada, inclusive os da vida pessoal, os da vida profissional-empresarial e os tantos outros feitos na academia. Estes desde a graduação, passando pelo mestrado e, nestes últimos anos, no processo de formação doutoral.

Segundamente, igualmente importante, o autor deste trabalho apresenta seus agradecimentos de ordem acadêmico-profissional. Tais agradecimentos se debutam inequivocamente aos orientadores, Jean Marcelo Simão e João Alberto Fabro que guiaram este trabalho e aportaram a reflexão, o espírito científico e o espírito filosófico, bem como, dentre outros, incutirão os sentimentos inestimáveis de resiliência e de colaboração. Neste sentido, um trabalho nunca é uma obra de um autor só, sempre há alguém que colabora de alguma forma sendo, mesmo que singelamente, um pouco coautor em alguma medida. No caso deles, entretanto, deve-se salientar que a orientação e colaboração foram veementes, inclusive para se alcançar a redação deste presente texto de documento de tese.

Ainda no âmbito acadêmico-profissional, particularmente na temática da tese proposta, seria absolutamente injusto não apresentar profundos agradecimentos a toda a equipe de pesquisa em torno do Paradigma Orientado a Notificações (PON), agradecimentos estes que se acentuam para todos aqueles que participaram da pesquisa e, em particular, os que participaram dos palcos de experimentação, seja na primeira fase ou na segunda fase dela. Em verdade, seria uma listagem considerável fazer nominalmente os agradecimentos e, talvez, em algo redundante dado que as menções e referências pertinentes a cada qual ocorrem ao longo do presente documento de tese.

Por fim, os agradecimentos de tese vão igualmente a cada membro da banca de avaliação deste trabalho. Inclusive, para a maioria dos membros, além de presente os agradecimentos também são retroativos, isto por terem já avaliado a qualificação de doutorado que ensejou o presente trabalho. Os agradecimentos presentes vão além de todo o esforço para avaliar este trabalho de pesquisa, à luz de todo conhecimento e sapiência de cada qual, mas também por o realizarem sobre um manuscrito longo, fato sobre o qual já se apresenta as devidas escusas. Entretanto, pertinente sublinhar que a natureza do trabalho exigira tal extensão, ainda que quiçá alguns conteúdos pudessem vir a se tornar apêndices e afins, tendo essa declaração seu coeficiente de mea-culpa.

Tudo isto posto, sinceros e profundos agradecimentos,

Adriano Francisco Ronszcka

RESUMO

RONSZCKA, Adriano Francisco. **Método para a Criação de Linguagens de Programação e Compiladores para o Paradigma Orientado a Notificações em Plataformas Distintas**. 2019. 375f. Tese de Doutorado – Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial (CPGEI). Universidade Tecnológica Federal do Paraná (UTFPR). Curitiba, 2019.

O Paradigma Orientado a Notificações (PON) tem sido explorado como uma alternativa promissora para a construção de sistemas computacionais. Em linhas gerais, as diretrizes do PON permitem desacoplar as expressões causais de programas. Isto se dá ao considerar cada uma dessas, bem como seus fatos relacionados, como entidades computacionais notificantes, possibilitando que o mecanismo de execução ocorra de forma reativa e naturalmente desacoplada. Com isso, o processo de inferência é isento de redundâncias. Tais redundâncias são comuns em outras abordagens de programação e, geralmente, afetam o desempenho de execução dos programas. Ainda, no modelo de notificações, o desacoplamento entre as entidades é intrínseco, o que facilita a construção de sistemas paralelos e/ou distribuídos. Ademais, por se orientar a regras, o PON tende a facilitar o desenvolvimento em alto nível. Devido a essas propriedades implícitas, o PON se destaca como paradigma emergente, necessitando, porém, de implementações sólidas e efetivas que as demonstrem conjuntamente. Embora alguns trabalhos tenham implementado materializações para o PON, tanto em software quanto em hardware, elas não contemplaram por completo as propriedades e características elementares do paradigma e, muitas vezes, se apresentaram de maneira incompleta e mesmo inconsistente. Além disso, outro fato agravante é a falta de padronização no processo de proposição e desenvolvimento de tais implementações, o que tende a dificultar a concepção de materializações efetivas. Nesse âmbito, esta tese propõe um método para a criação padronizada de materializações para o PON, denominado MCPON. Isto se dá, no método proposto, principalmente pela concepção e definição de linguagens de programação específicas, bem como pela implementação de compiladores próprios para o PON. Particularmente, estes artefatos são orientados por um elemento balizador na forma de um grafo diferenciado, denominado Grafo PON. Este mapeia os elementos de um programa e o fluxo execucional baseado em notificações do modelo computacional do PON. Com base no Grafo PON, o método permite a integração e compatibilidade entre as diferentes materializações construídas. Por fim, o método proposto foi aplicado por um grupo de desenvolvedores, os quais, ao passo em que adquiriram conhecimentos básicos sobre a construção de linguagens e compiladores, puderam aplicar etapas do método proposto em implementações sob plataformas distintas. Com base nisso, foi possível validar a pertinência do método proposto na tarefa de construção de materializações para o PON consistentes em si e entre si que respeitem as características e propriedades do paradigma tanto quanto permitir cada plataforma visada.

Palavras-chave: Paradigma Orientado a Notificações (PON). Linguagens de Programação e Compiladores. Método para a Criação de Linguagens e Compiladores para o PON.

ABSTRACT

RONSZCKA, Adriano Francisco. **Method for the creation of Programming Languages and Compilers for the Notification-Oriented Paradigm in Distinct Platforms**. 2019. 375f. Tese de Doutorado – Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial (CPGEI). Universidade Tecnológica Federal do Paraná (UTFPR). Curitiba, 2019.

The so-called Notification-Oriented Paradigm (NOP) has been explored as a promising alternative for the construction of computational systems. In short, the NOP guidelines allow decoupling programs' causal expressions. This is possible by considering each of these causal expressions and its related facts as notifying computational entities, thereby allowing the execution mechanism to occur in a reactive and naturally decoupled way. Actually, that allows performing an inference process free of redundancies. These redundancies are common in other programming approaches and generally affect the execution performance of programs. Moreover, the decoupling between the entities that compose the NOP notification model tends to facilitate the construction of highly parallel and/or distributed systems. In addition, the NOP is rule-oriented, which tends to facilitate high-level coding. Due to these implicit properties, the NOP stands out as an emerging paradigm, however demanding an effective implementation that can actually demonstrate them together. Even though there are several works implementing the NOP concepts, both in software and hardware, they do not fulfill all properties and features of the paradigm and, sometimes, they are incomplete and somehow inconsistent. In addition, another aggravating fact is the lack of standardization in the process of proposing and developing such implementations, which tends to make it difficult to conceive an effective implementation. In this context, this thesis proposes a method to guide the standardized creation of implementations for the NOP, called MCNOP. This is accomplished, in the proposed method, mainly by the design and definition of programming languages, as well as by the implementation of compilers for the NOP. Particularly, these artifacts are oriented by a central element in the form of a distinct graph, called NOP Graph. It maps the elements of a program and the execution flow based on notifications of the NOP computational model. The proposed method, based on the NOP Graph, allows the integration and compatibility between the different implementations conceived. Finally, the proposed method was applied by a group of developers that, after acquiring basic knowledge about language and compiler construction, were able to apply steps of the proposed method to implementations over distinct platforms. Based on that, it was possible to validate the pertinence of the proposed method for the creation of (individually and collectively) consistent implementation that respect the NOP features and properties as much as allowed by each targeted-platform.

Keywords: Notification-Oriented Paradigm (NOP). Programming Language and Compilers. Method for the Creation of NOP Languages and Compilers.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE CÓDIGOS

LISTA DE SIGLAS, ACRÔNIMOS E ABREVIATURAS

AFD	Autômatos Finitos Determinísticos
AFND	Autômatos Finitos Não-Determinísticos
API	*Application Programming Interface*
ArqPON	Arquitetura de Processador para o PON
AST	*Abstract Syntax Trees*
BNF	*Backus-Naur Form*
CON	Controle Orientado a Notificações
CoPON	Co-processador PON
CORBA	*Common Object Request Broker Architecture*
CPGEI	Pós-Graduação em Engenharia Elétrica e Informática Industrial
CRUD	*Create Retrieve Update Delete*
CTA	Controle de Trânsito Automatizado
DAG	*Direct Acyclic Graph*
DCG	*Definite Clause Grammar*
DOD	*Department of Defence*
DON	Desenvolvimento Orientado a Notificações
FBE	*Fact Base Element*
FIFO	*First-In First-Out*
FPGA	*Field-Programmable Gate Array*
FSF	*Free Software Foundation*
GCC	*GNU Compiler Collection*
IA	Inteligência Artificial
IC	Inteligência Computacional
IDE	*Integrated Development Environment*
IFC	Instituto Federal Catarinense
ION	Inferência Orientada a Notificações
IoT	*Internet of Things*
IP	*Internet Protocol*
ISA	*Instruction Set Architecture*
JIT	*Just-in-Time*
JSON	*JavaScript Object Notation*
LIFO	*Last-In First-Out*
LingPON	Linguagem do PON
LobeNOI	*Load balancing engine for NOI - Notification-Oriented Inference*
MCPON	Método de Compilação para o PON
MON	Metodologia de Projeto de Software Orientado a Notificações
NOCA	*Notification-Oriented Computer Architecture*
NOM	*Notification-Oriented Methodology*
NOP	*Notification-Oriented Paradigm*
NOPL	*Notification-Oriented Paradigm Language*
OO	Orientação a Objetos
PD	Paradigma Declarativo
PF	Paradigma Funcional
PI	Paradigma Imperativo
PL	Paradigma Lógico
POA	Paradigma Orientado a Agentes
POE	Paradigma Orientado a Eventos
PON	Paradigma Orientado a Notificações

PON-HD	PON em Hardware Digital
POO	Paradigma Orientado a Objetos
PP	Paradigma Procedimental
PPGCA	Programa de Pós-Graduação em Computação Aplicada
RAM	*Random Access Memory*
RdP	Redes de Petri
RI	Representação Intermediária
RMI	*Remote Method Invocation*
RNA	Redes Neurais Artificiais
RTL	*Register Transfer Language*
SBR	Sistemas Baseados em Regras
SO	Sistema Operacional
SRP	*Single-Responsibility Principle*
SSA	*Static Single Assignment*
STL	*Standard Template Library*
TCP	*Transmission Control Protocol*
UDP	*User Datagram Protocol*
UML	*Unified Modeling Language*
UTFPR	Universidade Tecnológica Federal do Paraná
VHDL	*VHSIC Hardware Description Language*
WAM	*Warren Abstract Machine*
XML	*eXtensible Markup Language*
YAML	*YAML Ain't Markup Language*

SUMÁRIO

CAPÍTULO 1

INTRODUÇÃO

Este trabalho de doutorado está inserido em um projeto de pesquisa sobre o Paradigma Orientado a Notificações (PON). O projeto em si é composto por uma base teórica e técnica com vários desdobramentos (SIMÃO, 2001; 2005; 2018). O presente trabalho, em especial, está focado na proposição de um método para criação de linguagens de programação e compiladores específicos para o PON com o intuito de permitir a concepção de materializações[1] efetivas para o paradigma, inclusive em plataformas distintas. Entende-se por "materialização efetiva para o PON" cada implementação do paradigma, seja em software ou em hardware, que contemple todas as características e conceitos do PON e que, principalmente, demonstre as propriedades elementares à luz de sua teoria.

De maneira geral, o PON apresenta algumas características relevantes, sendo que elas derivam de três propriedades elementares que o destacam como paradigma (SIMÃO, 2018):

- A primeira propriedade elementar do PON é a facilidade de desenvolvimento de software em alto nível com base na estruturação organizada dos elementos do modelo computacional do paradigma. Basicamente, tal modelo é composto por entidades facto-execucionais na forma de elementos notificantes e por entidades lógico-causais na forma de regras notificáveis (RONSZCKA *et al*., 2017).

- A segunda propriedade elementar do PON é a isenção de redundâncias estruturais (*i.e*., repetição de código) e temporais (*i.e*., reavaliação desnecessária de expressões), em virtude do modelo baseado em notificações pontuais. Isto permite inclusive a construção de programas com

[1] Em tempo, o termo "materialização" será utilizado ao longo desta tese para se referir a qualquer forma de implementação, seja em forma de códigos experimentais, *frameworks* ou, até mesmo, linguagens de programação e compiladores.

alta reatividade e, por consequência, com baixo tempo de processamento (BANASEWSKI, 2009; SIMÃO *et al.*, 2012a; RONSZCKA *et al*., 2015).

- A terceira propriedade elementar do PON é o desacoplamento explícito dos elementos que compõem o modelo (ou acoplamento mínimo, conforme o ponto de vista), devido a organização baseada em notificações. Isto viabiliza, dentre outros, a construção de programas com execução paralela ou distribuída tão fina quanto a arquitetura computacional permitir (PETERS *et al.,* 2012; LINHARES *et al.,* 2015; BELMONTE *et al*., 2016; BARRETO *et al*., 2018; OLIVEIRA *et al*., 2018; SCHÜTZ *et al.,* 2018; KERSCHBAUMER *et al*., 2018).

Nos dias atuais, tais características são desejáveis em linguagens e paradigmas de desenvolvimento de software, uma vez que a demanda por software otimizado, paralelizável e distribuível é crescente. Isto somado a “crise de software”, que se mantém ao longo da história, a qual indica a dificuldade frente ao rápido crescimento da demanda por software aliada à complexidade crescente dos problemas a serem resolvidos (BAUTSCH, 2007).

Nesse âmbito, este trabalho busca estruturar e facilitar a construção de materializações efetivas que demonstrem as propriedades elementares do PON de forma consistente, particularmente em relação à correta estruturação de suas características e conceitos. Além disso, este trabalho visa colaborar com a integração e consistência de boa parte dos atuais desdobramentos do projeto de pesquisa do PON, bem como prover uma base sólida para a construção de novas tecnologias que se baseiem no paradigma, visando inclusive plataformas computacionais distintas.

Neste capítulo introdutório, a Seção 1.1 contextualiza os fatores motivadores que justificam a preocupação com a forma com que as linguagens e paradigmas de programação são fundamentados e, além disso, apresenta o PON como alternativa aos paradigmas usuais. Ainda, tal seção apresenta as principais materializações do PON, bem como as lacunas que têm impedido a demonstração das propriedades elementares do PON de forma efetiva. A Seção 1.2, por sua vez, apresenta a justificativa para a necessidade da existência de um método para a criação de materializações padronizadas, coerentes e efetivas para o PON. A Seção 1.3

apresenta o objetivo principal e os objetivos específicos dessa tese. Por fim, a Seção 1.4 apresenta a organização dos capítulos subsequentes.

1.1 CONTEXTUALIZAÇÃO

Pesquisas realizadas nas últimas décadas contribuíram significativamente para a evolução tecnológica, sendo o setor de software fundamental para tal, dada sua quase onipresença hoje nos mais diversos campos de pesquisa e desenvolvimento tecnológico. Paralelamente e sinergicamente, a capacidade de processamento computacional cresceu igualmente em função da evolução das tecnologias (KEYES, 2006).

Historicamente, os avanços tecnológicos que resultaram no crescimento do desempenho dos processadores utilizados em sistemas computacionais dependeram, principalmente, do aumento da frequência de operação (*i.e., clock*) e do aumento da densidade de integração de semicondutores. Esse último fator, particularmente, continua a ser válido de acordo com os conceitos que fundamentam a Lei de Moore, segundo a qual a densidade de integração dobra a cada 24 meses (MOORE, 1965). Porém, a frequência do *clock* já não pode ser aumentada nas mesmas proporções históricas devido aos problemas de dissipação de potência, o que implica na necessidade de se explorar cada vez melhor o espaço disponível nos *chips* (GSCHWIND, 2006; BORKAR e CHIEN, 2011).

Nesse contexto, o aumento na disponibilidade de espaço para integração de circuitos mais complexos tem favorecido a replicação de múltiplos núcleos (*cores*) em uma mesma pastilha, resultando no chamado *multicore* e, por consequência, disseminando as chamadas arquiteturas paralelas (AGERWALA e CHATTERJEE, 2005). Embora, em tese, o aumento do número de unidades de processamento em paralelo permita aumentar o desempenho de execução da computação, na prática, isso depende de software que explore adequadamente o paralelismo.

O correto uso do paralelismo por software, por sua vez, depende de técnicas adequadas de desenvolvimento para tal. Entretanto, as técnicas de programação mais utilizadas atualmente para o desenvolvimento de software paralelo ou paralelizável, originadas da computação sequencial, normalmente apresentam problemas de acoplamento e redundância, além do modelo de programação naturalmente

sequencial em si (PROKOPEC, 2014; ASANOVIC *et al*., 2006; BORKAR e CHIEN, 2011).

Muito embora o setor de desenvolvimento de software tenha evoluído no tocante a permitir alguma programação visando paralelismo, tal evolução ainda não contempla de maneira efetiva o desenvolvimento de software que permita explorar naturalmente e efetivamente o paralelismo em software, salientando aqui ambientes como o *multicore* (ESCHMANN *et al.,* 2002). Ao bem da verdade, outras características como o real aproveitamento do processamento disponível, mesmo que não paralelo, ainda não são contempladas de maneira efetiva no desenvolvimento de software, incitando discussões sobre o assunto e, principalmente, motivando a proposição de novas técnicas para esse fim (ASANOVIC *et al*., 2006; BORKAR e CHIEN, 2011). Nesse âmbito, esse assunto é considerado na próxima subseção.

1.1.1 Paradigmas de programação

As técnicas de programação usuais fazem uso de linguagens de programação, as quais são regidas por paradigmas de programação. Nesse sentido, o nível de abstração de uma linguagem de programação está atrelado à essência de seus paradigmas. De uma maneira sucinta, pode-se considerar que existem dois grandes paradigmas de programação, nomeadamente o Paradigma Imperativo (PI) e o Paradigma Declarativo (PD), que se interseccionam em alguns aspectos, sendo, por vezes, o PD considerado uma abstração ou camada de alto nível sobre o PI (ROY e HARIDI, 2004; KAISLER, 2005; ROY, 2009; GABBRIELLI e MARTINI, 2010).

O PI pode ser dividido em dois outros paradigmas, o Paradigma Procedimental (PP) e Paradigma Orientado a Objetos (POO) sendo que o PP rege linguagens como C e Pascal e o POO linguagens como Java e C#, havendo hibridismos como no caso do C++. O PD, por sua vez, pode ser subdividido em outros dois paradigmas como o Paradigma Funcional (PF) e o Paradigma Lógico (PL) sendo que o PF rege linguagens como LISP (a qual possui um hibridismo com o PP) e que o PL rege linguagens como Prolog e OPS (ROY e HARIDI, 2004; KAISLER, 2005; ROY, 2009; GABBRIELLI e MARTINI, 2010). Na verdade, é usual que linguagens de programação tenham suas bases fundamentadas em mais de um (sub)paradigma, como os exemplos já citados. Além desses, linguagens de programação atuais como

Swift, Scala, Ruby, Python, Go e Lua também fazem uso de múltiplos paradigmas em sua essência.

Ainda que cada paradigma e subparadigma tenha suas peculiaridades, de maneira geral, linguagens de programação usuais oriundas deles não apresentam reais facilidades para a concepção de código cujos módulos tenham acoplamento mínimo e livres de redundâncias. Isso tende a afetar os custos de processamento, inviabilizar o reuso de módulos, dificultar a execução paralela e complicar a distribuição de processamento (GAUDIOT e SOHN, 1990; BANERJEE *et al.*, 1995; RAYMOND, 2003; SIMÃO e STADZISZ, 2009a; SIMÃO *et al.*, 2012a).

Esse quadro se dá devido à estrutura sequencial e a natureza de execução interdependente imposta pelos paradigmas que regem tais linguagens (HUGHES e HUGHES, 2003; BANASZEWSKI *et al.*, 2007; BANASZEWSKI, 2009). Em suma, os paradigmas de programação usuais, mais precisamente o PI e o PD, apresentam deficiências que afetam o desempenho das aplicações e, principalmente, a dificuldade na obtenção de desacoplamento (ou acoplamento mínimo) entre os módulos de software (BANASZEWSKI, 2009; SIMÃO e STADZISZ, 2009a; SIMÃO *et al.*, 2012a).

Essencialmente, o PI rege um modelo de construção sequencial de software orientado a percorrimentos ou buscas sobre elementos passivos, relacionando os dados (*e.g.*, variáveis e estruturas de dados) com expressões lógico-causais (*i.e.*, estruturas *if-then* ou similares). Na prática, a orientação a buscas implica na presença de redundâncias estruturais (*e.g.*, repetição de avaliações de uma mesma variável em partes distintas de um bloco de código) e redundâncias temporais (*e.g.*, reavaliação desnecessária de expressões que não tenham sido alteradas em um laço de repetição). Tais buscas afetam o desempenho das aplicações por gerar redundâncias de processamento e, principalmente, dificultam o alcance de uma dependência mínima entre os módulos pelo fato de gerar acoplamento implícito entre eles (GABBRIELLI e MARTINI, 2010; SIMÃO e STADZISZ, 2009a; SIMÃO *et al.*, 2012a).

O PD, por sua vez, apresenta-se como uma alternativa ao PI. Basicamente, o PD proporciona um nível maior de abstração, o que de certa forma, facilita a composição de programas amparados por tal paradigma (KAISLER, 2005; GABBRIELLI e MARTINI, 2010; SIMÃO *et al.*, 2012a). Além disso, algumas soluções declarativas evitam muitas das redundâncias de execução, a fim de otimizar o desempenho das aplicações, tais como os Sistemas Baseados em Regras (SBR) baseados nos algoritmos de inferência Rete (e seus derivados *TREAT*, *LEAPS*, *OPS*)

ou mesmo no assaz similar HAL. Ademais, as tecnologias baseadas no algoritmo Rete, como *Rule Works* e *ILog Rules*, apresentam importância na indústria (FORGY, 1982; MIRANKER e LOFASO, 1991; LEE e CHENG, 2002; KANG e CHENG, 2004; SIMÃO *et al.*, 2012a; RONSZCKA *et al*., 2015).

No entanto, programas construídos a partir de SBRs em geral (mesmo os baseados em Rete e HAL) também apresentam deficiências. Basicamente, tais soluções declarativas fazem uso de estruturas de dados, as quais são computacionalmente custosas, causando consideráveis sobrecargas de processamento. Assim, mesmo com código redundante, soluções baseadas no PI normalmente apresentam melhor desempenho do que as soluções baseadas no PD (BANASZEWSKI *et al.*, 2007; SIMÃO e STADZISZ, 2008; BANASZEWSKI, 2009; SIMÃO *et al.*, 2012a; RONSZCKA *et al.,* 2015, SCOTT, 2016).

Além disso, tal qual na programação baseada no PI, a programação no PD também gera acoplamento entre os módulos, uma vez que o processo de inferência também se baseia em buscas sobre entidades passivas (SIMÃO e STADZISZ, 2008; 2009a; GABBRIELLI e MARTINI, 2010; SIMÃO *et al.*, 2012a; RONSZCKA *et al.,* 2015). Ainda, outras abordagens como as baseadas em eventos e programação funcional, mesmo que amenizando alguns desses problemas, não os resolvem por completo, conforme literatura pertinente[2] (SIMÃO *et al.,* 2012; BROOKSHEAR, 2012; XAVIER, 2014; SCOTT, 2016).

Na realidade, ainda existem questões em aberto sobre o desenvolvimento de software*,* em termos de facilitadores para a composição de código não redundante e não acoplado (GABBRIELLI e MARTINI, 2010; SIMÃO et al., 2012a). Portanto, novas soluções, que simplifiquem a tarefa de construção de software com características que supram tais questões, são desejáveis e necessárias. Nesse contexto, a solução de programação chamada de Paradigma Orientado a Notificações (PON) foi proposta visando amenizar os problemas destacados (SIMÃO e STADZISZ, 2008; 2009a; BANASZEWSKI, 2009).

[2] Os principais paradigmas vigentes, nomeadamente PI e PD, são contextualizados, em maiores detalhes, no Capítulo 2, mais precisamente na Seção 2.1.

1.1.2 Paradigma Orientado a Notificações

Em linhas gerais, o chamado Paradigma Orientado a Notificações (PON) resolve, em certa medida, alguns dos problemas existentes nos paradigmas usuais de programação. Basicamente, o mecanismo de execução do PON previne a existência de redundâncias estruturais (*i.e*., repetição de código) e redundâncias temporais (*i.e*., reavaliação desnecessária de código), as quais são problemas comuns em outras abordagens de programação, conforme apresentado anteriormente e em mais detalhes no Capítulo 2.

O mecanismo de execução do PON unifica algumas das características e vantagens encontradas no PD (*i.e*., representação do conhecimento em regras) e do PI (*i.e*., flexibilidade de expressão e nível apropriado de abstração), procurando resolver, em termos de modelo, os problemas relatados anteriormente em aplicações de software, desde ambientes monoprocessados até os completamente multiprocessados ou distribuídos (SIMÃO e STADZISZ, 2008; 2009a; BANASZEWSKI, 2009; SIMÃO *et al.*, 2012a; BELMONTE *et al*., 2016; RONSZCKA *et al.,* 2017a; KERSCHBAUMER, 2018).

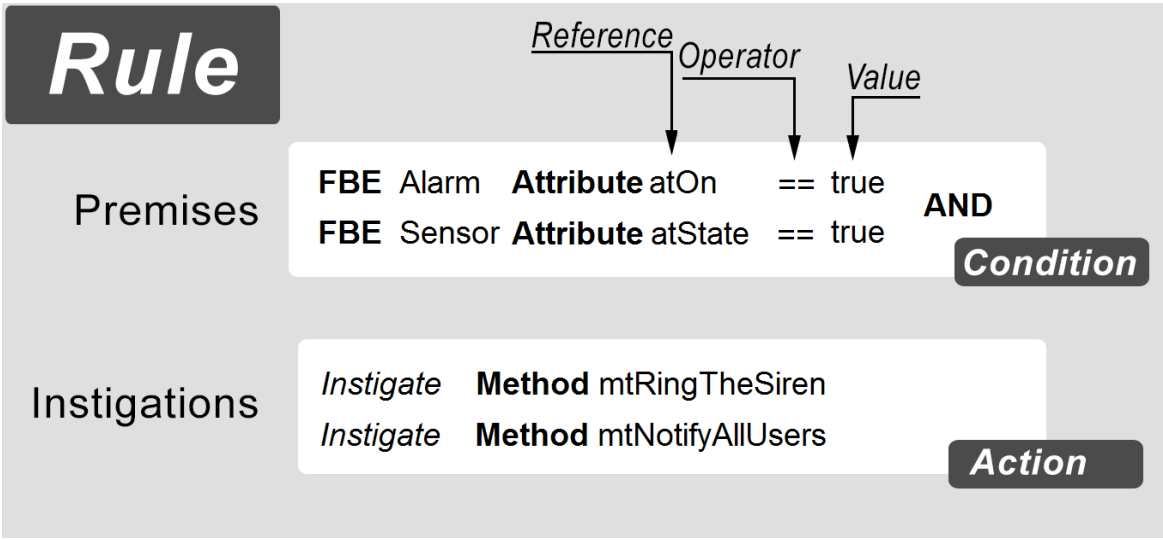
No PON há dois grupos de entidades de processamento. O primeiro grupo trata do processamento facto-execucional por meio de entidades notificantes chamadas de *Fact Base Elements* (*FBE*). O segundo grupo, por sua vez, trata do processamento lógico-causal por meio de entidades notificáveis chamadas de *Rules*. Tais entidades, seus constituintes e suas colaborações por notificações são objeto de explicação do texto que segue.

No PON, as instâncias de *Rule*, ou simplesmente *Rules*, gerenciam o conhecimento sobre qualquer comportamento lógico-causal no sistema. O conhecimento lógico-causal de cada *Rule* provém, normalmente, de regras “se-então”, as quais representam uma maneira natural de expressão desse tipo de conhecimento. Não obstante, esse conhecimento causal pode ser representado em outro formalismo equivalente quando se fizer pertinente, citando particularmente as chamadas Redes de Petri (RdP) com adaptações apropriadas (SIMÃO e STADZISZ, 2008; 2009a; BANASZEWSKI, 2009; WIECHETECK*,* 2011; WIECHETECK *et al.,* 2011; SIMÃO *et al.*, 2012a; KOSSOSKI, 2015, MENDONÇA, 2016).

A Figura 1 apresenta um exemplo de *Rule*, justamente na forma de uma regra causal. Uma *Rule* é composta por uma *Condition* e por uma *Action*, conforme mostra

a figura em questão. A *Condition* trata da decisão da *Rule*, enquanto a *Action* trata da execução das ações associadas. Assim sendo, *Condition* e *Action* trabalham para realizar o conhecimento causal da *Rule*. Na verdade, tanto a *Condition* quanto a *Action* são entidades computacionais agregadas à *Rule* (SIMÃO e STADZISZ, 2008; 2009a; BANASZEWSKI, 2009; SIMÃO *et al.*, 2012a).

Figura 1: Exemplo de uma *Rule*



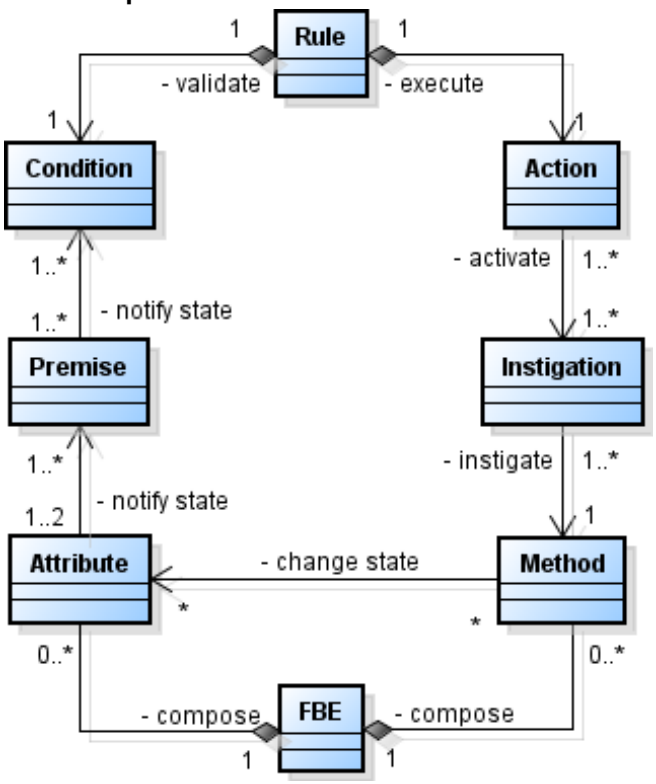
Fonte: Autoria Própria

A *Rule* apresentada na Figura 1 faria parte de um sistema de monitoramento de alarme, no qual a central de alarme e o sensor são tratados como entidades do sistema, mais precisamente como instâncias de *FBEs*. A *Condition* dessa *Rule* lida com a decisão de disparo do alarme, a qual é composta por duas *Premises* que se constituem em outro tipo de entidade computacional. Essas *Premises* em questão fazem as seguintes verificações sobre os *FBEs*: a) *o Alarme está ligado*? e b) *o Sensor detectou alguma anomalia*? Assim, por generalização, conclui-se que os estados dos 'atributos' dos *FBEs* compõem os fatos a serem avaliados pelas *Premises*.

Em tempo, os estados de cada um dos atributos de um *FBE* são tratados por meio de uma entidade chamada *Attribute*. Além do mais, e principalmente, para cada mudança de estado de um *Attribute* de um *FBE*, ocorrem subsequentes notificações e decorrentes avaliações lógicas somente e precisamente nas *Premises* relacionadas, conforme apresenta a Figura 2. Similarmente, a partir da mudança de estado das *Premises* (*e.g.*, uma avaliação lógica passar a ser verdadeira), ocorrem subsequentes notificações e decorrentes avaliações somente e precisamente nas *Conditions*

relacionadas com eventuais mudanças de seus estados (SIMÃO e STADZISZ, 2008; 2009a; BANASZEWSKI, 2009; SIMÃO *et al.*, 2012a).

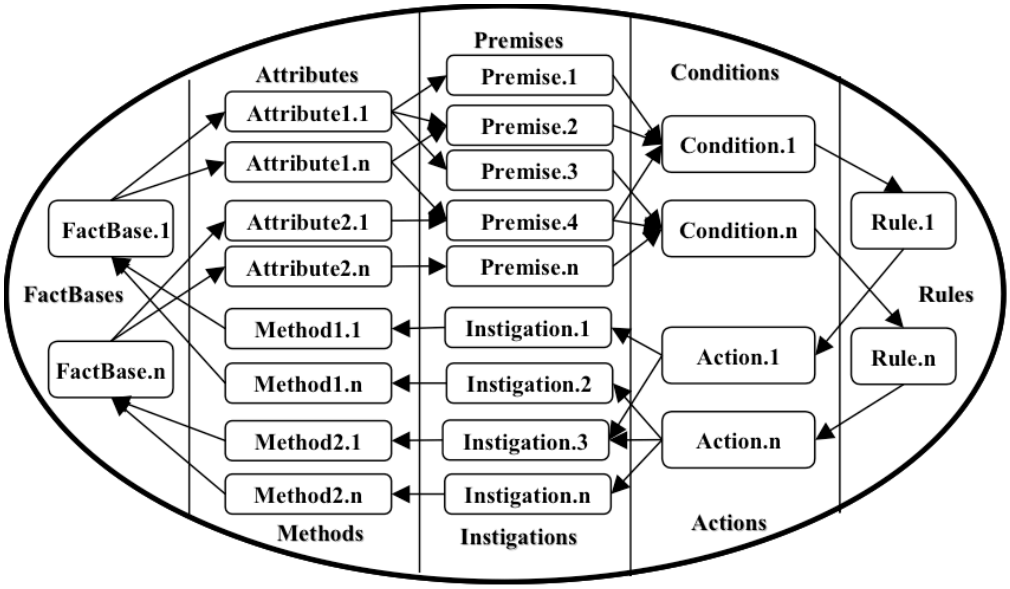
Figura 2: Principais entidades do PON e seus relacionamentos



Fonte: Ronszcka, 2012

Em suma, cada *Attribute* notifica as *Premises* relevantes sobre seus estados somente quando se fizer efetivamente necessário, normalmente quando houver mudança de valor. Cada *Premise*, por sua vez, notifica as *Conditions* relevantes dos seus estados usando o mesmo princípio. Baseado nesses estados notificados é que a *Condition* pode ser aprovada ou não. Se a *Condition* é aprovada, a respectiva *Rule* pode ser ativada executando sua *Action* (SIMÃO e STADZISZ, 2008; 2009a; BANASZEWSKI, 2009; SIMÃO *et al.*, 2012a). Isso tudo se dá por meio de uma cadeia de notificações entre entidades computacionais, conforme ilustra a Figura 3, o que se constitui no ponto central de inovação do PON.

Figura 3: Cadeia de Notificações



Fonte: Banaszewski, 2009

Em tempo, uma *Action* também é uma entidade computacional que se conecta a entidades computacionais de outro tipo, as *Instigations*. No exemplo dado, a *Action* contém duas *Instigations* para: a) *disparar a sirene*; e b) *notificar todos os usuários do sistema*. Efetivamente, o que cada *Instigation* faz é instigar (ativar) um ou mais métodos responsáveis por realizar serviços ou funcionalidades de um *FBE* (SIMÃO e STADZISZ, 2008; 2009a; BANASZEWSKI, 2009; SIMÃO *et al.*, 2012a).

Certamente, cada método de um *FBE* é também tratado por uma entidade computacional, que é chamada de *Method*. Geralmente, a execução de um *Method* muda o estado de um ou mais *Attributes*. Na verdade, os conceitos de *Attribute* e *Method* representam uma evolução dos conceitos de atributos e métodos de objetos do POO. A diferença fundamental é o desacoplamento implícito da entidade proprietária e a sua capacidade de notificação pontual para com *Premises* e *Instigations* nos termos explicados, o que seria uma "inteligência" colaborativo-notificante (SIMÃO e STADZISZ, 2008; 2009a; BANASZEWSKI, 2009; SIMÃO *et al.*, 2012a).

Outrossim, salienta-se que a ciência de qual elemento deve notificar outro se dá na própria composição das *Rules*, em tempo de construção do programa, o que poderia ser feito em um ambiente ou linguagem amigável na forma de regras causais. Em suma, cada vez que um elemento referenciar outro, o referenciado o consideraria como elemento a ser notificado quando houver mudanças em seu estado. Por exemplo, quando uma *Premise* faz menção a um dado *Attribute*, esse consideraria tal

Premise como elemento a ser notificado. Essa conexão por notificação entre as entidades que compõem cada programa é transparente para o desenvolvedor e, ao mesmo tempo, estabelece o mecanismo ou inferência por notificações particular a tais programas (SIMÃO e STADZISZ, 2008; 2009a; BANASZEWSKI, 2009; SIMÃO *et al.*, 2012a).

O mecanismo ou inferência por notificações é diferente do fluxo de iterações encontrado em aplicações do PI, incluindo o subparadigma OO, no qual o desenvolvedor informa de maneira explícita o laço de iteração por meio de comandos como *while* e *for*. Isto também é diferente do PD, no qual máquinas de inferência executam ciclos por meio de pesquisa ou buscas entre bases de fatos e bases de regras. No PON, a repetição ocorre de forma natural, na perspectiva de execução da aplicação, salientando que o fluxo de iterações das aplicações do PON é realizado de maneira transparente ao desenvolvedor, em virtude do orquestramento da cadeia de notificações pontuais entre as entidades notificantes (BANASZEWSKI, 2009; SIMÃO *et al.*, 2012a).

Outrossim, as diretrizes do PON permitem desacoplar as expressões causais do código-fonte, ao considerar cada uma dessas, bem como seus fatos relacionados, como entidades computacionais notificantes, possibilitando que o mecanismo de execução ocorra de forma reativa e naturalmente desacoplado. Em outras palavras, o fluxo de execução do PON acontece por meio das notificações pontuais entre suas entidades, as quais se apresentam com um nível de granularidade mínimo e com suas particularidades bem definidas. Isto permite que tais entidades orquestrem a execução de um programa de maneira objetiva e pontual, desacoplando as entidades e evitando buscas desnecessárias em grupos de entidades passivas, o que viabiliza inclusive paralelismo e distribuição fina de processamento (SIMÃO *et al.*, 2012a; RONSZCKA *et al.,* 2015a; RONSZCKA *et al.,* 2017).

A forma com que o PON trata expressões causais também é efetivamente diferente dos programas usuais do PI (salientando os Orientados a Objetos – OO) e do PD (salientado os chamados Sistemas Baseados em Regras – SBR). Nesses, as expressões causais são passivas e acopladas (senão fortemente acopladas) a outras partes do código, o que, de certa forma, dificulta o paralelismo ou a distribuição do processamento, bem como onera o desempenho do processamento (conforme o caso) (GABBRIELLI e MARTINI, 2010; SIMÃO et al., 2012a). Nesse sentido, a natureza do PON leva a uma nova maneira de compor sistemas, inicialmente em

software e subsequentemente em hardware digital, na qual os fluxos de execução são definidos por meio das entidades colaboradoras definidas no paradigma.

Em tempo, muito embora o PON permita compor sistemas em alto nível na forma de regras, esse conhecimento sobre a existência e *modus operandi* da colaboração por meio de notificações pode ser importante. Em certos contextos, como o de ambientes distribuídos, pode ser importante conhecer de antemão os impactos de desempenho de cada relação notificante, de maneira a agrupar as entidades com maior fluxo de notificações em um mesmo nó computacional (BELMONTE *et al*., 2016).

Com todos esses elementos considerados, o PON apresenta algumas propriedades elementares, vislumbradas em seu modelo teórico, que são: (a) facilidades com o desenvolvimento em alto nível com base na estruturação organizada dos elementos do modelo orientando-se a regras e entidades factuais; (b) ausência de redundâncias dada pela pontualidade das notificações, o que implica em baixo tempo de processamento; e (c) desacoplamento implícito dos elementos, o que viabiliza a execução paralela ou distribuída fina. Em suma, o PON representa uma nova maneira de estruturar, executar e pensar os artefatos de sistemas computacionais. Nesse sentido, algumas materializações do paradigma foram propostas e implementadas, ao longo dos anos, de modo a validar tais propriedades no âmbito do estado da técnica (BANASZEWSKI, 2009; VALENÇA, 2012; RONSZCKA, 2012; LINHARES, 2015; FERREIRA, 2015; KERSCHBAUMER, 2018).

1.1.3 Materializações dos conceitos fundamentais do PON

Diversos trabalhos do grupo de pesquisa do PON visaram implementar materializações que, de um modo ou de outro, contemplassem as propriedades elementares do PON, mesmo que de forma parcial. Nesse âmbito, as primeiras materializações do PON foram essencialmente em software. Atualmente, entretanto, existem tanto materializações do PON em software quanto em hardware digital (SIMÃO 2001; SIMÃO 2005; BANASZEWSKI, 2009; RONSZCKA, 2012; VALENÇA, 2012; FERREIRA, 2015). De fato, foi apenas em um segundo momento que surgiram as materializações em hardware digital (WITT *et al*., 2011; PETERS, 2012; LINHARES, 2015; KERSCHBAUMER, 2018).

Inicialmente, em software, foram criadas algumas materializações em forma de *frameworks*, o que possibilitou experimentar o fluxo de execução totalmente orientado a notificações, contemplando o modelo de entidades reativas, à luz da teoria do paradigma. De maneira geral, a primeira materialização para o PON, a versão prototipal do Framework PON C++, surgiu com o objetivo de validar o modelo computacional do PON, compondo regras de um sistema específico por meio das entidades notificantes do modelo conceitual do paradigma. Com isso, foi possível validar o mecanismo de execução baseado na reatividade das entidades notificantes do PON (SIMÃO, 2001; 2005, SIMÃO e STADZISZ, 2008).

Subsequentemente, na versão 1.0 do Framework PON C++, a estrutura da ferramenta permitiu criar programas de qualquer natureza, diferente da versão precedente que era mais limitada. Em termos de tempo de processamento, os resultados dos experimentos realizados evidenciaram o potencial do PON, principalmente em comparação a abordagens tradicionais do PD (*i.e.*, Rete e Rule Works). Conforme apresentado em (BANASZEWSKI, 2009; RONSZCKA *et al*., 2015), os programas construídos com base no Framework PON C++ 1.0 demonstraram desempenho superior nos casos de estudo realizados. Além disso, os mesmos programas apresentaram resultados razoáveis quando comparados a implementações baseadas no PI, ambos em linguagem C++ (BANASZEWSKI, 2009; SIMÃO *et al.*, 2012a).

Na versão 2.0 do Framework PON C++, por sua vez, houve a preocupação com a facilidade de programação, tanto na forma de criar programas em código, por meio de macros e um conjunto de padrões de implementação (RONSZCKA, 2012), quanto por meio de um software *Wizard* capaz de criar modelos (*templates*) de programas definidos de forma visual na ferramenta (VALENÇA, 2012). Além disso, a versão 2.0 do Framework PON C++ foi otimizada em termos de desempenho, apresentando, inclusive, estruturas de dados otimizadas para o processo de notificação (VALENÇA, 2012, SIMÃO *et al.,* 2017). Embora tenha alcançado resultados superiores em relação às versões precedentes, o tempo de processamento ainda não condizia com o esperado à luz do cálculo assintótico do PON (RONSZCKA *et al*., 2017a).

Na sequência, Belmonte *et al*. (2012) estenderam a implementação do *Framework* PON C++ 2.0, com o objetivo de paralelizar o fluxo de execução do PON em multinúcleo. Para isso, foi criado um conjunto de escalonadores de entidades,

definindo um escalonador para cada núcleo da máquina a ser utilizada, os quais escalonavam a execução do sistema por meio de *threads* independentes. Nesse contexto, a extensão do *framework* por meio de *threads*, intitulada de *Framework* PON C++ 3.0, permite ao desenvolvedor conceber software concorrente e também paralelo em ambiente *multicore* (BELMONTE *et al*., 2016).

Ainda, tal *framework* foi reestruturado de modo a tornar a implementação das entidades notificantes do PON mais desacopladas. Essa reestruturação possibilitou dividir a execução do programa de forma granular, permitindo inclusive que as entidades possam alternar livremente de núcleos de processamento em tempo de execução. Apesar de demonstrar a viabilidade do paralelismo de execução do PON com esta versão do *framework*, constatou-se que simplesmente dividir os elementos em núcleos não aumentava o desempenho de execução dos programas. Justamente o contrário, não raro o desempenho piorava em dezenas de vezes. Isso reforçou a necessidade de um método de balanceamento, especializado nas entidades do PON, de modo a organizar as entidades de acordo com suas conexões e frequência de notificações. Para isso, Belmonte *et al.,* (2016) propuseram o LobeNOI (*Load balancing engine for NOI – Notification-Oriented Inference – applications*) sobre o *Framework* PON C++ 3.0, justamente para realizar o balanceamento dinâmico da carga de trabalho de software PON em ambientes *multicore*.

Além disso, existem outras implementações prototipais de *frameworks* PON desenvolvidas em linguagem Java, C# e mesmo C++, porém com distribuição via TCP/IP (*Transmission Control Protocol/Internet Protocol*) (HENZEN, 2015; OLIVEIRA, 2016; TALAU, 2016; BARRETO, 2016; BARRETO *et al*., 2018, OLIVEIRA *et al.,* 2018), o que permitiu experimentar a factibilidade, mesmo que de forma preliminar, da distribuição das entidades do modelo em múltiplos nós de processamento. Mais recentemente, também foram desenvolvidos *frameworks* em AKKA e Erlang, buscando sinergia entre PON e o arcabouço tecnológico do chamado modelo de atores para fins de facilitação técnica de paralelismo e mesmo distribuição de entidades do PON (MARTINI, 2018b; NEGRINI, 2019).

De maneira geral, os *frameworks* permitiram a demonstração de factibilidades do PON, como a possibilidade efetiva de paralelismo e, até mesmo, distribuição, bem como a programação e desenvolvimento em alto nível, incluindo a criação de programas por meio de uma ferramenta visual de composição das regras - *Wizard* (VALENÇA, 2012). Entretanto, o custo computacional das estruturas de dados

utilizadas na concepção deles, mesmo na versão otimizada em C++, nomeadamente *Framework* PON C++ 2.0, não permitia alcançar todo o potencial do PON em termos de tempo de processamento, isto à luz de sua essência livre de redundâncias e de seu cálculo assintótico, o qual foi apresentado como sendo linear O(n) para o caso médio (SIMÃO, 2005; BANASZEWSKI, 2009; RONSZCKA *et al*., 2015).

Esse motivo, inclusive, levou a busca por novas materializações do PON em hardware digital (PETERS, 2012; LINHARES, 2015; KERSCHBAUMER, 2018). Existem, de fato, esforços para a materialização do PON em hardware, mais precisamente em hardware reconfigurável nomeados de *FPGAs* (*Field-Programmable Gate Array*). Uma primeira pesquisa e projeto, chamado de PON em Hardware Digital (PON-HD), faz uso de lógica reconfigurável por meio de *FPGAs*, visando explorar o seu potencial de paralelização (WITT *et al*., 2011; SIMÃO *et al*., 2012e; KERSCHBAUMER *et al*., 2015; PORDEUS *et al*., 2016; KERSCHBAUMER, 2018). Entretanto, a programação ocorreria em linguagem VHDL para o PON em hardware, o que demanda conhecimento específico para tal, fato este que fere uma das propriedades elementares do PON.

Ademais, em outra pesquisa, Peters (2012) propôs a implementação em lógica reconfigurável de um coprocessador PON (CoPON). Tal implementação se constitui em uma solução híbrida, na qual a parte da aplicação responsável pelo processamento factual é executada em um núcleo de processamento tradicional baseado na arquitetura conhecida como de “von Neumann”, usando uma adaptação do *Framework* PON C++ 1.0, e a parte da aplicação responsável pelo cálculo lógico-causal via propagação de notificações é executada por meio de um coprocessador, usando *FPGA*, baseado nos princípios do PON (PETERS, 2012; PORDEUS, 2017). Em tempo, a programação via *Framework* PON C++ 1.0 exige conhecimento técnico também.

Essas materializações demonstram, principalmente, a granularidade dos elementos do paradigma e promovem um fluxo de execução arquiteturalmente distinto aos criados em software, demonstrando a propriedade de alto paralelismo do paradigma. No entanto, as aplicações devem ser programadas em hardware digital, por meio de uma linguagem específica (*i.e*., *VHDL*), o que demanda conhecimento técnico especializado e, portanto, fere uma das propriedades elementares do PON, que é a facilidade de programação em alto nível. Ainda que Jasinski (2012) tenha feito um esforço inicial nesse âmbito, gerando hardware reconfigurável a partir de XML, a

questão permaneceu em aberto, dado que foi um protótipo limitado, servindo apenas como demonstração de conceito.

Ademais, outro problema na implementação de hardware reconfigurável é que para cada nova aplicação que surge, uma nova reconfiguração do hardware deve ser realizada (Linhares, 2015). Nesse sentido, foi desenvolvido por Linhares (2015) uma arquitetura de computação própria e um processador baseado nos princípios do PON, o chamado ARQPON em português ou *NOCA* (*Notification-Oriented Computer Architecture*) em inglês (LINHARES *et al.,* 2014; 2015). O intuito desse processador é prover sempre o mesmo hardware independente da aplicação.

Ainda, inspirado nos processadores tradicionais, o ARQPON provê uma linguagem *Assembly-like* para realizar sua programação, na verdade um *Assembly-PON* (LINHARES *et al.,* 2014; 2015; LINHARES, 2015). Subsequentemente, também foi criado um simulador do ARQPON, que dentre outros, demonstrou a escalabilidade da arquitetura, contornando limitações de espaço nas *FPGAs* disponíveis pelo grupo de pesquisa do PON, até então (PORDEUS, 2017). Entretanto, tanto na arquitetura quanto no simulador, a programação ocorreria em linguagem *Assembly-PON*, o que demanda conhecimento específico para tal, fato este que fere uma das propriedades elementares do PON.

1.2 MOTIVAÇÃO

O fato do PON possuir as características e propriedades aqui apresentadas é instigante dado que elas permitiriam atender demandas contemporâneas e históricas no desenvolvimento de software. Uma das demandas é justamente a concepção de linguagens/paradigmas de programação de mais alto nível e amigáveis, com processamento mais performante, bem como uso apropriado de paralelismo ou distribuição nas arquiteturas modernas de computação, como *multicore* e *manycore*. Este tipo de solução permitiria mitigar, em alguma medida, a chamada ‘crise do software’, na qual se estabelece que a demanda por software será efetivamente maior do que a capacidade de produzi-los.

Em função da necessidade de soluções novas e apropriadas no tocante ao desenvolvimento de software, o PON deveria ser efetivamente materializado, visando assim, ser uma contribuição apropriada neste âmbito dado. Entretanto, embora tenha havido pertinentes avanços nas materializações do PON, que vão mesmo além do

software chegando ao hardware digital, observou-se que as materializações construídas até então não proporcionam resultados de todo satisfatórios. Isto se dá justamente por tais materializações não contemplarem efetivamente uma ou outra propriedade elementar do próprio PON.

Em suma, as materializações do PON em software via *frameworks* até são capazes de prover alguma facilidade de programação de software e mesmo paralelismo em multinúcleo, mas não conseguem atingir um tempo de processamento adequado em cada núcleo/processador. Isto se dá principalmente pelas limitações da construção de tais ferramentas, as quais normalmente se baseiam por estruturas de dados computacionalmente custosas. Por outro lado, as materializações do PON em hardware digital apresentam, em alguns casos, tempo de processamento satisfatório e também paralelismo intrínseco, mas isso em detrimento da facilidade de programação ou desenvolvimento.

1.3 JUSTIFICATIVA

O fato concreto é que o PON ainda não alcançou todas as propriedades e características vislumbradas no seu modelo. Sendo assim, ainda existe a necessidade de evoluções no estado da arte, bem como no estado da técnica, para que o PON atingisse o que fora vislumbrado em sua concepção, principalmente no âmbito de suas propriedades elementares. Para isso, as materializações do PON deveriam apresentar, cada qual, facilidades com a programação em alto nível, processamento temporalmente e estruturalmente não redundante, levando ao baixo tempo de processamento (*i.e.*, alta reatividade) e à estruturação desacoplada de seus entes, permitindo a computação com paralelismo ou distribuição implícita tanto quanto a arquitetura computacional visada permitir.

Nesse âmbito, primeiramente vislumbrou-se a possibilidade da construção de uma linguagem e compilador próprios para o PON visando contemplar todas as propriedades elementares do PON de forma conjunta, tanto quanto possível na arquitetura visada. Inicialmente ou prototipalmente, isto se deu em arquitetura Von Neumann monoprocessada, atendendo aos requisitos de programação orientada a regras em alto nível à luz da natureza do PON (RONSZCKA *et al.*, 2013). A construção deste compilador preliminar, todavia, apresentou algumas dificuldades no decorrer de

seu desenvolvimento, uma vez que as técnicas e métodos de compilação tradicionais não se adequam por completo às características do PON (RONSZCKA *et al*., 2013).

De maneira geral, as técnicas e métodos de compilação utilizados nos paradigmas usuais, nomeadamente PI e PD, não seriam apropriados para o PON justamente por ele ser um novo paradigma. Enquanto em compiladores convencionais as entidades computacionais compõem árvores de encadeamentos de comandos e identificadores, os quais são normalmente relacionados por meio de uma estrutura algorítmica, no PON, idealmente, o mapeamento de um programa deveria se dar por meio de um grafo de entidades conectadas por meio de suas colaborações por notificações. Isto já se observou de pronto no primeiro protótipo do compilador, o que exigiu o advento de uma nova técnica no tocante a compilação para o PON (RONSZCKA *et al*., 2013).

Naturalmente, esta carência se relaciona fortemente ao fato de inexistir um método de compilação adequado às características do PON que permita criar linguagens e compiladores para o paradigma de maneira efetiva. Além disso, tal método deveria manter a coerência de linguagens e compiladores entre si e, ao mesmo tempo, explorar as propriedades do PON, tanto quanto possível em cada plataforma visada. Outro problema é que cada linguagem e compilador criado para o PON teria que garantir suas propriedades elementares de forma a manter a coerência entre as materializações (*targets*) associadas. Essa coerência significa, em suma, que um mesmo programa feito em uma ou mais linguagens de programação próprias ao PON teriam que executar de maneira equivalente após a compilação por diferentes compiladores próprios ao PON, mesmo que em plataformas distintas.

Isto posto e levando tudo o que foi descrito em consideração, conclui-se que o PON ainda carece de materializações efetivas que de fato contemplem conjuntamente suas propriedades elementares de forma efetiva e coerente, em um ambiente próprio de programação e de execução estável. Em suma, a inexistência de uma materialização efetiva para o PON está fortemente relacionada a falta de um método apropriado que norteie a criação padronizada de tais materializações. Esse fato criou, até então, certa barreira entre as soluções propostas, dificultando a evolução individual de cada qual, principalmente pela falta de coerência entre elas, impedindo, inclusive, o potencial de integração entre elas. Isto se explica e se agrava justamente pela falta de conceitos de compilação que estejam harmonizados com a teoria do PON.

Nesse sentido, esta tese surge ao encontro dessas necessidades e se propõe saná-las, de maneira a propor um método para a criação de materializações padronizadas e coerentes para o PON com base em uma nova forma de compilação própria, com a introdução de conceitos próprios ao PON. Particularmente, o método baliza a definição de linguagens de programação específicas para o PON, bem como define a construção de compiladores especializados na geração de código para plataformas distintas que se baseariam no fluxo de execução orientado a notificações.

Assim, as contribuições deste trabalho devem responder os seguintes questionamentos:

- É possível criar um conjunto de conceitos e técnicas de compilação apropriadas para o PON?
- É possível criar materializações para o PON de forma padronizada e consistente?
- É possível integrar as colaborações de diversos desenvolvedores de linguagens e compiladores do PON em um único sistema de compilação voltado para plataformas distintas?
- É possível validar as materializações criadas em relação aos conceitos implementados?

As pesquisas apresentadas neste trabalho buscam responder essas questões, apresentando modelos e exemplos desenvolvidos para esse fim. Ademais, é importante salientar que o método proposto foi extensivamente validado por meio de experimentações técnico-práticas, bem como foi amplamente explorado por diversos pesquisadores que trabalharam tanto na sua implementação e evolução quanto na experimentação das linguagens e compiladores gerados. Esta colaboração sinérgica entre diversos pesquisadores permitiu o amadurecimento do método e das tecnologias envolvidas, as quais são detalhadas ao longo deste trabalho.

1.4 OBJETIVOS

O objetivo geral desta tese é propor um método de construção de linguagens e compiladores consistentes para o Paradigma Orientado a Notificações (PON), permitindo contemplar suas propriedades elementares em plataformas distintas.

Para atingir esse objetivo, este trabalho apresenta os seguintes objetivos específicos:

- Estudar as particularidades das linguagens de programação e compiladores de diversos paradigmas e plataformas, bem como as principais técnicas e métodos de compilação existentes, a fim de estabelecer elementos reaproveitáveis, assim como identificar as lacunas existentes relativas à natureza do PON.
- Estabelecer um conjunto de conceitos e técnicas de compilação para o PON, à luz da lacuna existente, de forma tal que este sirva de base para a construção de um método próprio ao PON no tocante a linguagens e compiladores.
- Estabelecer o método de construção de linguagens e compiladores para o PON, de forma que ele se configure à luz da natureza deste paradigma emergente, visando principalmente a padronização e coerência entre diversas materializações decorrentes deste método, bem como a exploração apropriada das propriedades elementares e características do PON, tanto quanto possível em cada plataforma visada.
- Implementar o método proposto com as suas etapas, por meio do reaproveitamento de tecnologias pertinentes e implementação do necessário, no tocante a conceitos e técnicas que preenchem lacuna existente, formando assim um sistema de compilação (*i.e.,* a materialização do método em questão).
- Criar ao menos uma linguagem e um conjunto de compiladores próprios para o PON com base nas etapas do método proposto, por meio do seu sistema de compilação, com o intuito de validá-los.
- Disponibilizar o método e o seu respectivo sistema de compilação para outrem (especialistas e não-especialistas em PON) para aplicar as etapas do método proposto na prática, inclusive em plataformas distintas, com o intuito de revalidá-los.
- Avaliar os resultados obtidos no âmbito da aplicação do método e seu sistema de compilação na construção de linguagens e compiladores para o PON em plataformas distintas, inclusive no tocante a sua integralidade.

1.5 ORGANIZAÇÃO DO TRABALHO

A sequência deste trabalho está descrita em seis capítulos[3]. No Capítulo 2 são apresentados os temas fundamentais que formam a base conceitual utilizada para o desenvolvimento deste trabalho. O capítulo apresenta sucintamente os paradigmas de programação usuais e suas derivações, bem como sua relação íntima com as linguagens de programação regidas por estes. Ainda, este capítulo apresenta os principais conceitos relacionados a teoria de linguagens e construção de compiladores.

No Capítulo 3, a seu turno, são apresentados os conceitos do PON de forma detalhada, bem como uma descrição aprofundada de suas propriedades elementares. Além disso, o capítulo apresenta a evolução das diversas materializações construídas ao longo dos últimos anos, bem como a plenitude de cada qual em relação aos conceitos implementados e propriedades elementares demonstradas no tocante ao próprio PON.

No Capítulo 4, particularmente, é apresentado a proposta desta tese, o método de compilação para o PON denominado de MCPON. Este capítulo apresenta em detalhes as particularidades do método, o qual foi especificamente arquitetado para a criação de linguagens e compiladores particulares ao PON. Ademais, o capítulo apresenta as raízes e objetivos do método, bem como as etapas necessárias para a aplicação do MCPON na construção de linguagens e compiladores próprios para o PON.

No Capítulo 5, por sua vez, é apresentada a aplicação do método proposto, ainda que em sua versão preliminar (ou *alfa*), no âmbito da criação da LingPON. Este capítulo é particularmente importante para a contextualização histórica do MCPON, uma vez que o amadurecimento do método em si foi fortemente amparado pelo acúmulo de experiências em torno da implementação da LingPON e de seus compiladores especializados. O capítulo também explica as evoluções da LingPON à luz do MCPON, ao longo dos anos de doutoramento, particularmente amparadas pelas padronizações definidas pelo método proposto. Ademais, o capítulo apresenta

[3] A título de informação, caso o leitor apresente conhecimento sobre os capítulos, poderá não os ler ou postergar a leitura, sem prejuízo ao entendimento deste trabalho. Isto, em especial, ao Capítulos 2 (fundamentação teórica) e ao Capítulo 3 (Revisão do PON).

a história da LingPON, contextualizando as motivações para sua criação. Além disso, o capítulo explora as evoluções da LingPON no âmbito de compilação e geração de código, especialmente voltadas a plataformas distintas, com o objetivo de construir materializações efetivas e que contemplem, tanto quanto possível, as propriedades elementares do PON.

No Capítulo 6, a seu turno, é apresentado estudos, no qual são contempladas e demonstradas todas as etapas do método proposto, nomeadamente MCPON, de maneira mais precisa. Para isso, foi desenvolvida uma nova linguagem para o PON, denominada de NOPL e também conhecida por LingPON 2.0, especialmente voltada para resolver as principais questões em aberto na versão anterior da linguagem. Além disso, este estudo contou com a participação de um grupo de desenvolvedores, os quais contribuíram com a construção desta nova linguagem e, em especial, na construção de um conjunto de geradores de código voltados especificamente para plataformas distintas. Ademais, o capítulo apresenta a avaliação de pertinência do método, com base nos resultados provenientes dos trabalhos desenvolvidos por este grupo de desenvolvedores.

Por fim, o Capítulo 7 apresenta as conclusões sobre o trabalho desenvolvido, assim como as perspectivas dos possíveis desdobramentos em trabalhos futuros.

CAPÍTULO 2

REFERENCIAL TEÓRICO

Este capítulo apresenta os principais conceitos para estruturar esta tese. Primeiramente, a Seção 2.1 apresenta reflexões sobre o atual estado da arte dos paradigmas de programação usuais na computação. A Seção 2.2, por sua vez, apresenta a teoria de linguagens e de compiladores, à luz do estado da arte e da técnica, bem como as etapas essências do processo de compilação. Por fim, a Seção 2.3 apresenta as considerações finais deste capítulo[4].

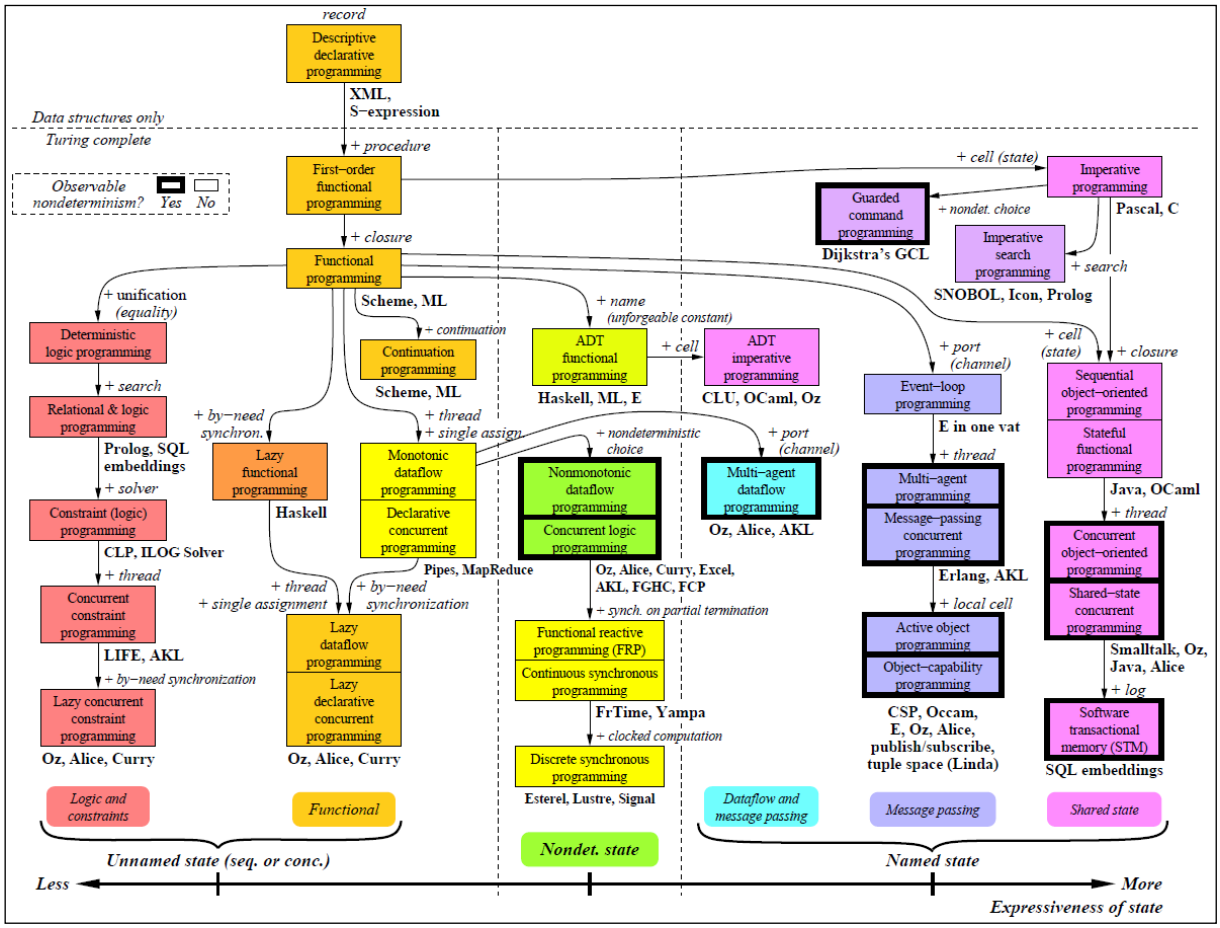
2.1 PARADIGMAS DE PROGRAMAÇÃO

Na ciência da computação, o termo “paradigma” é empregado como uma maneira de abstrair o pensamento do programador em uma determinada estrutura computacional capaz de definir o fluxo de execução de um programa. De acordo com David Watt (2004), os paradigmas se diferem em conceitos e abstrações utilizadas para representar os elementos de um programa (*e.g.*, objetos, funções, variáveis e restrições) e a maneira com que esses interagem de maneira a ditar o fluxo de execução de tal programa (*e.g.*, atribuições, avaliações causais, repetições, empilhamento e recursividade) (WATT, 2004).

Segundo Peter Van Roy (2009), um paradigma de programação é um sistema formal que define como a programação é realizada. Ademais, cada paradigma tem o seu conjunto de técnicas e conceitos de programação que, conjuntamente, definem sua forma de estruturar o pensamento na construção de programas (VAN ROY, 2009). Peter Van Roy desenvolveu uma taxonomia para paradigmas de programação, conforme apresenta a Figura 4 (VAN ROY, 2009).

[4] A título de informação, caso o leitor apresente conhecimento sobre as seções deste capítulo, poderá não as ler ou postergar a leitura, sem prejuízo ao entendimento desta tese.

Figura 4: Taxonomia de Paradigmas de Programação



Fonte: Van Roy, 2009

Conforme apresenta a Figura 4, os paradigmas de programação são organizados em um grafo que basicamente apresenta o relacionamento conceitual entre eles. Ao todo, são 27 quadros, cada qual representando um paradigma e seu respectivo conjunto de conceitos. Ademais, setas entre dois quadros representam a adição de novos conceitos, no sentido de que os quadros derivados, contemplam os conceitos dos paradigmas anteriores, acrescidos de um ou mais novos conceitos que, conjuntamente, os definem como um paradigma distinto dos demais (VAN ROY, 2009).

Os conceitos são basicamente elementos primitivos básicos que, em conjunto, dão origem aos paradigmas. Com frequência, dois paradigmas que se apresentam de formas assaz distintas diferem por apenas um único conceito, como por exemplo o Paradigma Funcional que é composto pelos conceitos de registros, procedimentos e escopo assim como também o Paradigma Orientado a Objetos é composto, com exceção de que esse último também é composto por estados nomeados com alta expressividade (VAN ROY, 2009). Além disso, a Figura 4 destaca

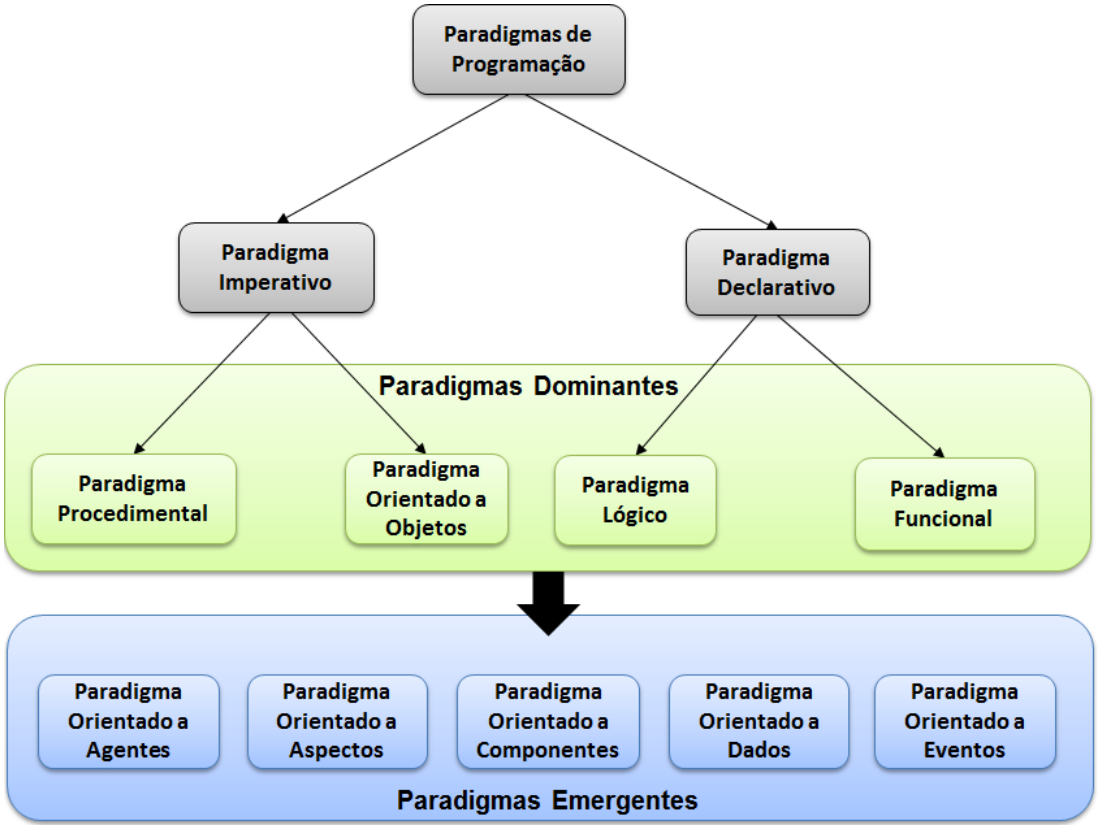
a presença de duas propriedades importantes nos paradigmas, se eles apresentam, ou não, o não-determinismo (ou indeterminismo) observável e quão fortemente eles suportam estados nomeados.

Em linhas gerais, o indeterminismo observável é expresso na figura por meio de bordas espessas. Na prática, o indeterminismo observável ocorre quando a execução de um programa não é completamente determinada por sua especificação, ou seja, em algum momento durante a execução de dado programa, o controle do fluxo de execução será determinado pelo programa em *runtime*. O indeterminismo é observável se o mesmo programa gerar resultados diferentes para as mesmas entradas. A tendência de paradigmas que apresentam indeterminismo observável é que a programação se torna mais complexa, como é o caso da Programação Orientada a Objetos Concorrente. Em contrapartida, a Programação Declarativa Concorrente é totalmente determinística, o que tende a facilitar a programação (VAN ROY, 2009).

A segunda propriedade de um paradigma é o quão fortemente ele suporta estados nomeados. Basicamente, o conceito de estado nomeado é a habilidade de um programa de reter informação ou, mais precisamente, de armazenar uma sequência de valores em memória (*e.g.,* variáveis, vetores e mapas). Na Figura 4 os eixos são divididos em três níveis de expressividade, variando de estados não-nomeados (mais a esquerda) a estados nomeados (mais a direita), determinísticos e indeterministicos, bem como sequencial ou concorrente. Totalizando 8 combinações possíveis (VAN ROY, 2009).

Apesar da taxonomia de Van Roy apresentar relevância e definir os paradigmas de acordo com seus conceitos fundamentais, estes podem ser classificados de uma forma simplificada. Como apresentado por alguns autores (BANASZEWSKI, 2009; GABRIELLI e MARTINI, 2010; BROOKSHEAR, 2012, SEBESTA, 2012), os paradigmas de programação usuais (dominantes e emergentes) poderiam ser classificados como subconjuntos de dois paradigmas maiores, o Paradigma Imperativo (PI) e o Paradigma Declarativo (PD). A Figura 5 ilustra a classificação desses paradigmas de uma maneira mais resumida e simplificada em relação à Figura 4.

Figura 5: Classificação dos paradigmas de programação



Fonte: Ronszcka, 2012

Conforme ilustra a Figura 5, o PI pode ser entendido como constituído pelo Paradigma Procedimental (PP) e pelo Paradigma Orientado a Objetos (POO), os quais se diferenciam essencialmente na forma como os elementos e instruções são representados e organizados, sendo o POO considerado mais rico e supostamente estruturado em termos de expressão do código. O PD, por sua vez, pode ser entendido como constituído essencialmente pelo Paradigma Lógico (PL) e pelo Paradigma Funcional (PF).

Em tempo, conforme o ponto de vista, PP, POO, PL e PF poderiam ser considerados paradigmas e não subparadigmas ainda que com características mais próximas uns dos outros. Em todo caso, tais paradigmas se enquadram na primeira camada, à luz da Figura 5, pertencente ao grupo dos paradigmas dominantes que orienta a construção das linguagens de programação vigentes (BROOKSHEAR, 2012; SCOTT, 2016). Outrossim, não raro as linguagens de programação contemplam mais de um (sub)paradigma vigente, como C++ que é procedimental e orientada a objetos. Há mesmo linguagens de intersecção como LISP que possui aspectos lógico-declarativos e outros imperativo-procedimentais (VAN ROY, 2009).

Ainda, no tocante à segunda camada de paradigmas que representam os paradigmas emergentes, há outros hibridismos. Geralmente, os paradigmas de programação emergentes são estabelecidos por meio de um arquétipo ou *framework* em uma materialização de um paradigma dominante, o qual forma uma camada intermediária entre os conceitos do paradigma emergente e o paradigma dominante. Ademais, os paradigmas emergentes, em alguns casos, podem ser implementados sob os princípios de diferentes paradigmas dominantes. Um exemplo seria o caso do Paradigma Orientado a Agentes que tem sido implementado por abordagens imperativas e declarativas. Os paradigmas emergentes também podem ser objeto de implementação multiparadigma híbrida (BANASZEWSKI, 2009; HANSEN e FOSSUM, 2010; XAVIER *et al*., 2014).

Em todo caso, em linhas gerais, tanto o PI quanto o PD apresentam similaridades ao serem baseados em buscas/percorrimentos sobre entidades passivas, as quais consistem em dados (*e.g*., fatos ou estados de variáveis ou de atributos de outras entidades computacionais) e comandos de decisão (*e.g*., expressões causais como *se-então* ou regras). Tais buscas afetam o desempenho das aplicações por gerar redundâncias de processamento e acoplamento implícito entre as entidades que compõem uma aplicação (BANASZEWSKI *et al.,* 2007; SIMÃO e STADZISZ, 2008; 2009a; SIMÃO *et al.*, 2012a).

Essencialmente, o PI impõe pesquisas orientadas a laços de repetições sobre elementos passivos, relacionando os dados (*i.e*., variáveis, vetores e árvores) a expressões causais (*i.e*., se-então ou declarações similares). Tais relacionamentos normalmente impactam negativamente no desempenho de tais aplicações, devido sua estrutura monolítica, prolixa e acoplada, o que gera a execução de código não-otimizado e interdependente (BANASZEWSKI *et al.,* 2007; SIMÃO e STADZISZ, 2008; GABBRIELLI e MARTINI, 2010; BROOKSHEAR, 2012; SIMÃO *et al.*, 2012a).

Por sua vez, essencialmente, o PD permite um nível maior de abstração e, portanto, maior facilidade de programação (KAISLER, 2005; GABBRIELLI e MARTINI, 2010). Além disso, algumas soluções declarativas podem evitar muitas das redundâncias de execução, a fim de otimizar o processamento. Dentre tais soluções, citam-se os Sistemas Baseados em Regras (SBRs), com base em algoritmos de inferência otimizados como o HAL ou o industrialmente conhecido Rete (FORGY, 1982; CHENG e CHEN, 2000; LEE e CHENG, 2002; KANG e CHENG, 2004). No entanto, programas construídos com base em linguagens de programação usuais do

PD (*e.g.*, *LISP*, *PROLOG* e SBRs em geral) ou mesmo construídos com base em soluções otimizadas (*e.g.*, SBRs baseados no algoritmo Rete) também apresentam desvantagens (BANASZEWSKI *et al.,* 2007; SIMÃO e STADZISZ, 2008; SIMÃO *et al.*, 2012a).

As soluções do PD são compostas por estruturas de dados de alto nível, as quais são normalmente dispendiosas em termos de processamento. Isso, de fato, agrega custos de processamento consideráveis, impactando diretamente no desempenho das aplicações. Assim, mesmo com a presença de código redundante, as soluções do PI são normalmente melhores em desempenho do que as soluções do PD (BANASZEWSKI, 2009; SCOTT, 2016).

Além disso, similarmente à programação no PI, a programação no PD também gera acoplamento forte entre os módulos. Isto devido ao processo de inferência ser também baseado em pesquisa sobre entidades passivas (SIMÃO e STADZISZ, 2008; 2009a; GABBRIELLI e MARTINI, 2010). Ainda, outras abordagens entre o PI e o PD, tais como a programação dirigida por eventos ou a programação funcional, não resolvem tais problemas. Em alguns casos, essas até reduzem, atenuam ou fatoram tais problemas, mas não efetivamente os resolvem (BROOKSHEAR, 2012; SIMÃO *et al.*, 2012a; SCOTT, 2016).

2.1.1 Reflexões Pontuais da Programação Imperativa

As principais desvantagens da Programação Imperativa estão voltadas para a redundância de código e acoplamento (SIMÃO e STADZISZ, 2009a). A primeira afeta o tempo de processamento e também, tal qual a segunda, o processo de desacoplamento. Isto, por sua vez, afeta o processo de reaproveitamento de módulos/partes e o processo de distribuição de processamento. Estas questões são detalhadas nas subseções seguintes.

2.1.1.1 Redundâncias

Na Programação Imperativa, incluindo a Programação Orientada a Objetos (POO), a presença de código redundante e interdependente é resultado da maneira com a qual as expressões causais são organizadas e consequentemente avaliadas. Isso é exemplificado no Código 1 que representa um código habitual e elaborado sem

grande esforço técnico e intelectual no POO. Outramente dito, isso significa que o código foi elaborado de uma forma não complicada, como idealmente as aplicações deveriam ser concebidas (SIMÃO e STADZISZ, 2008; BANASZEWSKI, 2009; GABRIELLI e MARTINI, 2010; BROOKSHEAR, 2012).

Código 1: Exemplo de código redundante na Programação Imperativa

```
1  . . .
2  while (true) do
3    if ((object_1.attribute_1 = 1) and
4        (object_2.attribute_2 = 1) and
5        (object_3.attribute_3 = 1))
6    then
7         object_1.method_1();
8         object_2.method_1();
9         object_3.method_1();
10   end_if
11   . . .
12   if ((object_1.attribute_1 = 1) and
13       (object_2.attribute_n = n) and
14       (object_3.attribute_n = n))
15   then
16        object_1.method_n();
17        object_2.method_n();
18        object_3.method_n();
19   end_if
20 end_while
21 . . .
```

Fonte: Simão *et al.*, 2012a

Nesse exemplo, é observado que o laço de repetição força a avaliação de todas as condições (ou inferência decorrente) de maneira sequencial. No entanto, a maioria das avaliações é desnecessária, uma vez que normalmente somente alguns atributos apresentam alterações em seus estados em cada iteração (SIMÃO *et al.,* 2012a).

Esse tipo de algoritmo apresentado até pode ser considerado não importante nesse exemplo simples e pedagógico, sobretudo se o número (n) de expressões causais for pequeno. Entretanto, se for considerado um sistema complexo, integrando muitas partes como aquela, pode-se ter uma grande diferença de desempenho de processamento em função das redundâncias. Em tempo, esse tipo de código apresenta pelo menos dois tipos de redundâncias, nomeadamente a temporal e a estrutural (PAN *et al.*, 1998; SIMÃO e STADZISZ, 2008; 2009a; SIMÃO *et al.*, 2012a).

A redundância estrutural ocorre quando uma expressão lógica não é compartilhada ou, mais precisamente, quando seu valor booleano não é compartilhado entre outras expressões causais pertinentes, causando reavaliações desnecessárias. A redundância temporal, por sua vez, ocorre quando uma avaliação

lógico-causal é realizada repetidas vezes sobre um elemento já avaliado e inalterado. De fato, ambos os tipos de redundâncias estão presentes no código exemplo apresentado, o qual poderia ser otimizado via esforço de programação adicional. Entretanto, em escala, isto tornaria a programação no PI mais difícil, o que se constitui em outro problema. Efetivamente, a dificuldade de programação é reconhecidamente uma das dificuldades no desenvolvimento de software em PI (KAISLER, 2005; SIMÃO e STADZISZ, 2008; 2009a; BANASZEWSKI, 2009; VAN ROY, 2009; GABRIELLI e MARTINI, 2010).

2.1.1.2 Acoplamento

Além das usuais avaliações repetitivas e desnecessárias no código imperativo, os elementos avaliados em expressões causais são passivos, embora eles sejam essenciais nesse processo, em todo o caso. Por exemplo, uma dada declaração *if-then* ou se-então (ou seja, uma expressão causal) e as variáveis (ou seja, elementos avaliados) não tomam parte na decisão com relação ao momento em que devem ser avaliados. Em tempo, quem força tal momento de avaliação é o laço de repetição que impera sobre eles (SIMÃO e STADZISZ, 2008; 2009a; GABRIELLI e MARTINI, 2010; SIMÃO *et al.*, 2012a).

De fato, as avaliações de tais expressões causais e seus respectivos elementos são realizadas sequencialmente pela linha de execução principal (ou pelo menos em linhas de execução presente em cada *thread*) de um programa, comumente guiada por meio de um conjunto de laços de repetição. Como essas expressões causais e seus respectivos elementos não conduzem ativamente sua própria execução (ou seja, eles são passivos), a sua interdependência não é efetivamente explícita em cada execução do programa (SIMÃO e STADZISZ, 2009a; GABRIELLI e MARTINI, 2010; SIMÃO *et al.*, 2012a).

Assim, as expressões causais e seus respectivos elementos avaliados, dependem dos resultados das avaliações ou estados de outros elementos, quando não uns dos outros. Isso significa que eles são acoplados de alguma maneira e que deveriam ser dispostos conjuntamente, pelo menos, no contexto de cada módulo para terem alguma coerência e coesão. Esse acoplamento aumenta a complexidade do código, dificultando, por exemplo, (futuro) reaproveitamento de partes do código ou (eventual) paralelização/distribuição do código, sendo tão mais difícil quão mais fina

for tal paralelização ou distribuição. Isso faz com que cada módulo, ou até mesmo o programa como um todo, seja entendido como uma entidade computacional monolítica (KAISLER, 2005; SIMÃO e STADZISZ, 2009a; GABRIELLI e MARTINI, 2010; SIMÃO *et al.*, 2012a).

2.1.1.3 Dificuldade de Distribuição

Mais particularmente, quando a distribuição (*e.g.*, distribuição de processo, processador ou distribuição por *cluster*) é pretendida, uma análise de código poderia identificar um conjunto de elementos menos dependente, definindo módulos mais fidedignos, o que facilitaria, portanto, sua posterior divisão e distribuição. No entanto, tal atividade é normalmente complexa devido ao acoplamento existente no código e a complexidade resultante da programação imperativa (BANERJEE *et al.,* 1995; WACHTER *et al.*, 2004; SIMÃO *et al.*, 2012a).

Nesse sentido, um software bem construído, composto por módulos minimamente acoplados, amparado por artefatos de engenharia de software, como programação orientada a aspectos ou a aplicação do projeto axiomático via técnicas apropriadas (SEVILLA *et al.*, 2008; PIMENTEL e STADZISZ, 2006), poderia ajudar no processo de distribuição (SIMÃO *et al.*, 2012a). Ainda, *middlewares* como CORBA e RMI poderiam ser úteis em termos de infraestrutura para alguns tipos de distribuição, caso exista um desacoplamento suficiente entre os módulos de software (AHMED, 1998; REILLY e REILLY, 2002; SEVILLA *et al.*, 2008; SIMÃO *et al.*, 2012a).

Apesar desses avanços, a distribuição de cada elemento de código ou, até mesmo, de cada módulo de código, ainda é uma atividade complexa, exigindo outros esforços de pesquisa (WACHTER *et al.*, 2004; GAUDIOT e SOHN, 1990; TILEVICH e SMARAGDAKIS, 2002; JOHNSTON *et al.*, 2004; SEVILLA *et al.*, 2008). Nesse âmbito, ainda seriam necessários esforços adicionais para alcançar facilidade de distribuição (*e.g.*, distribuição automática, rápida e em tempo real), incluindo melhorias no processo de distribuição como um todo (*e.g.*, distribuição balanceada e minimamente interdependente) (SIMÃO e STADZISZ, 2009a; SIMÃO *et al.*, 2012a).

A dificuldade de distribuição é um problema, uma vez que existem contextos em que a distribuição é realmente necessária (COULOURIS *et al.*, 2001; HUGHES e HUGHES, 2003; GRUVER, 2007). Por exemplo, um dado programa otimizado e que, ainda assim, excederia a capacidade de um processador disponível, poderia ter seu

processamento dividido em um conjunto de processadores (OLIVEIRA e STEWART, 2006). Tais características podem ser encontradas em diversas aplicações, citando algumas como planta nuclear (DÍAZ *et al.*, 2007), manufatura inteligente (DEEN, 2003; SIMÃO, 2005; TIANFIELD, 2007; SIMÃO, TACLA e STADZISZ, 2009) e controle cooperativo (KUMAR *et al.*, 2005; SIMÃO *et al.*, 2012a).

Além disso, existem outras aplicações que são inerentemente distribuídas e precisam de uma distribuição flexível, tais como a computação ubíqua motivada pela Internet das Coisas (Internet of Things – IoT – do inglês) (FERSI, 2015). Exemplos mais precisos são das redes de sensores e algum controle de produção inteligente (LOKE, 2006; TIANFIELD, 2007), ou mais recentemente dos *smart objects*, como relógios, lâmpadas, termostatos, bicicletas, torradeiras etc., os quais, em conjunto, criam um sistema complexo de objetos conectados na internet (BANDYOPADHYAY e SEN, 2011).

Assim, em suma, a facilidade e a correta distribuição seriam efetivamente esperadas, uma vez que existe uma crescente redução de preços no mercado de processadores (AI IMPACTS, 2015) e também existe, inclusive, consideráveis avanços nos modelos de comunicação em rede (TANENBAUM e STEEN, 2002; BANASZEWSKI, 2009; BANDYOPADHYAY e SEN, 2011; SIMÃO *et al.*, 2012a).

2.1.1.4 Dificuldade de Desenvolvimento

Além das questões de otimização e problemas de distribuição, o desenvolvimento de programas com a Programação Imperativa (PI) pode ser visto como difícil devido sua sintaxe complicada e uma diversidade de conceitos a serem aprendidos, tais como: ponteiros, variáveis de controle e laços aninhados (GIARRATANO e RILEY, 1993).

O processo de desenvolvimento seria propenso a erros, uma vez que quase todo o código é realizado de forma manual, com base em tais conceitos. Neste contexto, o algoritmo imperativo exemplificado (Algoritmo 3) poderia certamente ser otimizado, no entanto sem facilidades significativas, ainda mais quando sua essência de *n* expressões causais (*e.g.*, comando se-então) avaliando *m* elementos factuais (*e.g.*, variáveis) estiver dispersa em um sistema de maior porte (SIMÃO *et al.*, 2012a).

Seria necessário investigar soluções melhores do que aquelas fornecidas pelo PI. Em um primeiro olhar, a solução para resolver alguns de seus problemas poderia

ser o uso de linguagens de programação de outros paradigmas, como a Programação Declarativa, a qual automatiza o processo de avaliação de expressões causais e seus elementos (RUSSEL e NORVIG, 2003; ROY e HARIDI, 2004; SIMÃO *et al.*, 2012a). Ainda, essa abordagem proporciona abstrações que minimizam a realização de algumas tarefas (*i.e*., programa-se "o que fazer" ao invés de "como fazer") (BANASZEWSKI, 2009). Entretanto, o PD também tem os seus poréns.

2.1.2 Reflexões pontuais da Programação Declarativa

Um exemplo bem conhecido da Programação Declarativa e sua natureza é um Sistema Baseado em Regras (SBR) (GIARRATANO e RILEY, 1993; SIMÃO e STADZISZ, 2009a). Os SBRs provêm uma programação em alto nível baseado na composição de regras lógico-causais, o que minimiza o contato dos desenvolvedores com particularidades algorítmicas (GIARRATANO e RILEY, 1993). Os SBRs são compostos por três entidades modulares gerais, nomeadamente Base de Fatos, Base de Regras e Motor de Inferência, as quais possuem responsabilidades distintas. Na verdade, essa forma é usual e geral em linguagens declarativas ou com elementos declarativos (*e.g*., PROLOG e CLIPS) (RUSSEL e NORVIG, 2003; SIMÃO *et al.*, 2012a).

Na Programação Declarativa, os estados das variáveis são tratados em uma Base de Fatos e o conhecimento causal em uma Base Lógico-Causal (ou Base de Regras na programação SBR), as quais são automaticamente combinadas por meio de um Motor de Inferência (GIARRATANO e RILEY, 1993; KANG e CHENG, 2004). Além disso, alguns algoritmos de inferência, como Rete (FORGY, 1982; CHENG e CHEN, 2000; LEE e CHENG, 2002; KANG e CHENG, 2004), TREAT (MIRANKER, 1987; MIRANKER e LOFASO, 1991), LEAPS (MIRANKER *et al.*, 1990) e HAL (LEE e CHENG, 2002), evitam considerável parte das redundâncias temporais e estruturais (BANASZEWSKI, 2009). No entanto, as estruturas de dados utilizadas para resolver esses problemas implicam em muito consumo de capacidade de processamento (FORGY, 1982; SIMÃO, *et. al*., 2012).

Além disso, em geral, um motor de inferência relacionado a uma determinada linguagem declarativa tende a limitar a criatividade do desenvolvedor, o que dificulta algumas otimizações algorítmicas e obscurece o acesso ao hardware, o que pode ser

inadequado em determinados contextos (WATT, 2004; BANASZEWSKI, 2009; BROOKSHEAR, 2012; SIMÃO *et al.*, 2012a; SCOTT, 2016).

A solução para esses problemas pode ser a mescla entre a programação Declarativa e a Imperativa (ROY e HARIDI, 2004; WATT, 2004). Na verdade, tal abordagem foi proposta em soluções como *CLIPS*++ (GIARRATANO e RILEY, 1993) e *ILOG Rules* (ALBERT, 1994; ILOG, 1998). No entanto, tais soluções não são populares devido a fatores como a mistura de sintaxe, mistura de paradigmas e razões técnico-culturais (BANASZEWSKI, 2009). De qualquer forma, mesmo a Programação Declarativa se apresentando como uma solução relevante, ela não resolve todos os problemas (SIMÃO e STADZISZ, 2008; 2009a; SIMÃO *et al.,* 2012a).

De fato, além da sobrecarga de processamento, a Programação Declarativa também apresenta acoplamento em seu código. Similarmente à Programação Imperativa, programas declarativos possuem um fluxo de execução ou política de inferência, cuja essência é uma entidade monolítica, *i.e*., uma máquina ou motor de inferência. Tal política de inferência é responsável por, de maneira centralizadora, analisar todos os dados passivos (base de fatos) e inferir a partir do estado desses a aprovação ou reprovação das expressões lógico-causais (regras) afetadas por tais estados. Assim, a inferência baseada em técnica de pesquisa (*i.e*., *matching*) implica em forte dependência entre a base de fatos e a base de regras (SIMÃO e STADZISZ, 2009a; SIMÃO *et al.*, 2012a).

2.1.3 Reflexões pontuais sobre outras abordagens de programação

Melhorias no contexto do PI e do PD têm sido aplicadas com o intuito de reduzir os efeitos de códigos baseados em pesquisas redundantes, tais como a Programação Orientada a Eventos (POE) e a Programação Funcional (PF) (RUSSEL e NORVIG, 2003; FAISON, 2006; BANASZEWSKI, 2009). A POE e a PF têm sido usadas na concepção de diferentes tipos de software, como controle discreto, interfaces gráficas, sistemas de atores e sistemas multiagentes (RUSSEL e NORVIG, 2003; FAISON, 2006; BANASZEWSKI, 2009; SIMÃO *et al.*, 2012a).

Essencialmente, na POE, cada evento (*e.g*., um botão pressionado, uma interrupção de hardware ou uma mensagem recebida) desencadeia uma dada execução (*e.g*., procedimento, processo ou método), geralmente em um tipo determinado de módulo (*e.g*., bloco, objeto ou, até mesmo, ator/agente), ao invés de

análises sucessivas e repetidas das expressões condicionais para a sua execução. O mesmo princípio se aplica à chamada PF, cuja diferença estaria nas chamadas de funções através de outras funções, em substituição aos eventos. Ainda, função nesse contexto significaria procedimento, método ou alguma unidade similar. Outrossim, programação funcional e orientada a eventos utilizadas em conjunto seria algo usual (FAISON, 2006; BROOKSHEAR, 2012; SEBESTA, 2012; XAVIER, 2014).

No entanto, o algoritmo em cada processo, método ou função acionado por eventos ou chamada de função é constituído essencialmente usando a Programação Declarativa ou Imperativa. Isso implica nas deficiências encontradas nesses estilos de programação, como redundância de código, acoplamento e afins, ainda que fatorado em alguma medida. De fato, se cada módulo possuir uma considerável quantidade de código causal, eles podem se apresentar como um problema quando em conjunto, tanto em termos de mau uso de processamento quanto em termos de dificuldade de distribuição (SIMÃO *et al.*, 2012a, BROOKSHEAR, 2012; XAVIER, 2014; SCOTT, 2016).

Outrossim, uma abordagem alternativa de programação é a chamada Programação Orientada a Fluxo de Dados (JOHNSTON *et al.*, 2004), que supostamente deveria permitir a execução do programa orientada por dados, em vez de uma linha de execução com base na pesquisa sobre os dados. Portanto, isso facilitaria o desacoplamento e a distribuição (JOHNSTON *et al.*, 2004; SIMÃO *et al.,* 2012a).

Na verdade, a distribuição da Programação Dirigida por Fluxo de Dados é obtida no processamento aritmético, porém não é realmente alcançada em processamento do tipo lógico-causal (GAUDIOT e SOHN, 1990; JOHNSTON *et al.*, 2004). No caso de processamento lógico-causal em *data-flow* para plataformas usuais (baseadas em von Neumann), tal processamento seria realizado por programação funcional ou, senão, por intermédio de motores de inferência, tais como Rete, HAL e afins (GAUDIOT e SOHN, 1990; TUTTLE e EICK, 1992; SIMÃO *et al.*, 2012a).

O fato é que os motores de inferência atuais tentam alcançar uma abordagem orientada a fluxo de dados. No entanto, o processo de inferência ainda se baseia em pesquisas, mesmo que se utilizando de assaz otimizadas “árvores” ou grafos de dados. Sendo assim, os problemas relatados persistem (SIMÃO *et al.*, 2012a; RONSZCKA *et al.,* 2015).

2.1.4 Linguagens de Programação

De maneira geral, o modelo de um paradigma está disponível para o desenvolvedor por meio de uma linguagem de programação, sendo que, em um primeiro momento, as primeiras linguagens criaram o paradigma imperativo e, subsequentemente, o paradigma imperativo-procedimental. Depois, novos paradigmas foram criados, como o Paradigma Funcional (PF), Paradigma Lógico (PL) e Paradigma Orientado a Objetos (POO) com subsequentes ensejos de linguagens efetivas para expressá-los mais veementemente (SEBESTA, 2012).

Nesse âmbito, algumas linguagens de programação foram desenvolvidas para suportar principalmente seus (sub)paradigmas dominantes, apesar de não-raro também suportarem outros paradigmas. A título de exemplo, linguagens como Java e Smalltalk suportam principalmente o POO, Haskell e Standard ML são baseados no PF e Prolog e Mercury são suportados pelo PL. Ainda, há outras linguagens que suportam múltiplos paradigmas em sua essência, como C++, Leda e Oz (ROY e HARIDI, 2004). Esse tipo de linguagem é denominado de multiparadigmas.

Uma linguagem de programação multiparadigmas é uma linguagem que provê um arcabouço no qual os programadores podem trabalhar com uma variedade maior de estilos, inter-relacionando estruturas de diferentes paradigmas. O principal objetivo de uma linguagem de programação multiparadigmas é proporcionar aos desenvolvedores uma ferramenta mais flexível, uma vez que nenhum paradigma por si só fornece a melhor solução para todos os problemas. Entretanto, a utilidade de uma linguagem multiparadigmas depende de quão bem os diferentes paradigmas estão integrados (ROY e HARIDI, 2004).

De maneira a apresentar as características e particularidades das linguagens de programação, o Apêndice A deste documento foi elaborado especificamente para apresentar uma breve história das linguagens de programação e, portanto, de seus paradigmas, bem como apresentar as principais linguagens utilizadas nos dias atuais. Ademais, tal apêndice também apresenta algumas características para a avaliação de linguagens de programação em relação a sua possível popularidade.

2.2 TEORIA DE LINGUAGENS E COMPILADORES

Nos primórdios da computação e por muitos anos, os primeiros softwares criados foram escritos de forma manual em código de máquina e posteriormente em linguagem *Assembly* e em notação matemática. As linguagens de alto nível de programação não foram inventadas até que os benefícios da reutilização de software em diferentes CPUs compensassem o esforço de se escrever um compilador. Naquela época, a capacidade de memória dos primeiros computadores era muito limitada, o que dificultava a criação de linguagens de alto nível e, principalmente, compiladores (WEXELBLAT, 1981).

No final da década de 1950, as linguagens de programação independentes de máquina começaram a surgir, bem como vários compiladores experimentais associados. O primeiro compilador foi escrito por Grace Hopper em 1952 (HOPPER, 1952; LEMONE, 1992), para a linguagem de programação A-0 (WEXELBLAT, 1981). Entretanto, tal compilador se apresentava mais um *loader* ou *linker* do que um compilador completo propriamente dito. Em geral, a equipe de desenvolvimento do Fortran liderada por John Backus na IBM é geralmente creditada como tendo introduzido o primeiro compilador completo em 1957, embora tenha ocorrido simultaneamente ao desenvolvimento do Algebraic Translator de Laning e Zierler (WEXELBLAT, 1981).

Ademais, naquela época, surgiram as primeiras teorias para a construção de linguagens e compiladores, as quais ainda têm sido aplicadas atualmente na construção de compiladores usuais. Em tempo, as características e propriedades das linguagens de programação, bem como o impacto de seus (sub)paradigmas, influenciaram diretamente na criação dos compiladores tradicionais, principalmente em relação a essência imperativa e orientada a pesquisas sob elementos passivos, comum a tais paradigmas, especialmente por influência da arquitetura Von Neumann.

De modo a apresentar em maiores detalhes as particularidades da construção de linguagens de programação e compiladores, a Seção 2.2.1 apresenta a teoria de linguagens formais e a Seção 2.2.2 apresenta a teoria de construção de compiladores.

2.2.1 Teoria das linguagens formais

Na década de 1950 originou-se a teoria das linguagens formais com o objetivo de desenvolver teorias relacionadas com as linguagens naturais. A Teoria das linguagens formais é o estudo de modelos matemáticos que possibilitam a especificação e o reconhecimento de linguagens (no sentido amplo da palavra), suas classificações, estruturas, propriedades, características e inter-relacionamentos. De maneira geral, a estrutura de uma linguagem de programação pode ser amplamente especificada por meio de um formalismo denominado gramática. Uma gramática é escrita em uma linguagem de descrição de linguagens, ou metalinguagem, e seu propósito é definir todas as *strings* (*i.e.,* cadeias de caracteres) permitidas que puderem formar um programa correto (TUCKER e NOONAN, 2010). As regras gramaticais definem a gramática a ser utilizada na linguagem construída, compondo assim a estrutura da linguagem.

O linguista Noam Chomsky criou uma classificação para gramáticas formais, chamando-a de Hierarquia de Chomsky. Esta classificação possui quatro níveis (GRUNE *et al*, 2012):

- Nível 0 - linguagens irrestritas ou recursivamente enumeráveis;
- Nível 1 - linguagens sensíveis ao contexto;
- Nível 2 - linguagens livres do contexto;
- Nível 3 - linguagens regulares.

Ainda, segundo Grune *et al*. (2012), os dois últimos níveis (os níveis 2 e 3) são amplamente utilizados na descrição de linguagens de programação e na implementação de interpretadores e compiladores, portanto discutiremos apenas esses dois níveis ao longo desta tese.

As linguagens de nível 3, mais especificamente, as linguagens regulares são linguagens formais que podem ser expressas a partir de expressões regulares. De acordo com a hierarquia de Chomsky, linguagens regulares são aquelas geradas por gramáticas regulares. Basicamente, uma linguagem regular pode ser definida como uma linguagem reconhecida por um autômato finito. Um autômato finito é basicamente uma máquina de estados finita capaz de definir se uma cadeia de caracteres representa uma sentença válida em uma dada linguagem. A palavra finito é incluída

no nome para ressaltar que um autômato só pode conter uma quantidade limitada de estados e de transições entre esses (AHO, 2008).

Um autômato finito pode ser representado diagramaticamente por um grafo dirigido e rotulado, chamado de diagrama de transições, no qual os nós são os estados e as transições rotuladas com os símbolos do alfabeto de entrada, determinam o próximo estado em função do estado atual e do símbolo de entrada. Há um estado inicial, chamado de estado de partida, e um ou mais estados, denominados estados finais, que indicam onde o autômato deve parar no processo de reconhecimento para que uma sequência de símbolos de entrada seja aceita (AHO, 2008).

Ademais, essa diagramação pode ser usada para representar tanto autômatos finitos não-determinísticos (AFND) quanto autômatos finitos determinísticos (AFD). Em um AFND, mais de uma transição de estado pode ser possível para o mesmo símbolo de entrada a partir de um único estado. Já em um AFD existe no máximo uma transição de estado definida para cada símbolo a partir de um único estado. Os dois são capazes de reconhecer precisamente linguagens simples classificadas como regulares. Na prática, os autômatos finitos são os mais simples reconhecedores de linguagens (AHO, 2008).

Por sua vez, nas linguagens de nível 2 segundo a hierarquia de Chomsky, as gramáticas livres de contexto normalmente são utilizadas na etapa de análise sintática de uma linguagem de programação. Tal gramática se apresenta como um formalismo que descreve a estrutura de programas na forma de uma linguagem de programação (Grune *et al*., 2012). Ademais, a gramática livre de contexto consiste em símbolos terminais, símbolos não-terminais, um símbolo inicial e produções.

De maneira geral, símbolos terminais são os símbolos básicos a partir dos quais as cadeias de caracteres são formadas (*e.g.,* palavras reservadas e demais *tokens*). Os símbolos não-terminais são variáveis sintáticas que representam conjuntos de cadeias. Os conjuntos de cadeias não-terminais auxiliam na definição da linguagem gerada pela gramática. Os não-terminais impõem uma estrutura hierárquica sobre a linguagem que é a chave para a construção de um analisador sintático (FERREIRA, 2015).

Também componente da gramática, uma produção, por sua vez, é um par contendo um símbolo não-terminal e uma cadeia de símbolos terminais e não-terminais. Assim, produções são consideradas como regras, sendo o símbolo não-terminal como lado esquerdo da regra e a cadeia de símbolos, o lado direito. A

definição de uma produção inicia com a definição de um símbolo inicial (*start*). Esse símbolo é substituído pelo lado direito de uma das suas regras. Tais substituições são efetuadas para todas as regras de produção, até a identificação de uma sequência de caracteres contendo apenas símbolos terminais. Assim, as produções de uma gramática especificam a forma como os terminais e os não-terminais podem ser combinados para formar cadeias. Nesse âmbito, o Código 2 apresenta um código de exemplo para uma gramática livre de contexto.

Código 2: Exemplo de gramática livre de contexto

```
1  <S>            ::= <if_clause>;
2
3  <if_clause>    ::= IF <exp> THEN <code_block> END
4                    | IF <exp> THEN <code_block> ELSE <code_block> END
5
6  <exp>          ::= ID <fator> ID
7
8  <code_block>   ::= RETURN NUMBER
9
10 <fator>        ::= EQ
11                   | NE
```

Fonte: Aho *et al.*, 2008

No Código 2, mais especificamente na linha 1, a definição da gramática inicia com um *token* principal, ou seja, um símbolo inicial referenciado pelo símbolo não-terminal *S*. A este símbolo não-terminal está associada uma regra gramatical com dois símbolos: o não-terminal *if_clause*, e o terminal "*;*". O não-terminal *if_clause*, definido na linha 3, tem a ele associado duas regras gramaticais (duas produções). A diferença entre as produções é a possibilidade de construir um *if_clause* com uma estrutura simples ou com uma condicional extra (*i.e.,* com o bloco *ELSE)*. Cada símbolo não-terminal apresenta sua própria regra e produção neste modelo, como exemplo, o símbolo não-terminal "*exp*" que é utilizado para avaliar uma expressão dentro da cláusula "*if_clause*".

Em tempo, é válido ressaltar, que a forma utilizada para descrever o exemplo do Código 2 é a Forma de Backus-Naur (abreviada como *BNF*, de *Backus-Naur Form*). Em 1960, a BNF foi adaptada da teoria de Chomsky por John Backus e Peter Naur para expressar uma definição sintática formal para a linguagem de programação Algol. Desde então, a BNF tem sido amplamente utilizada como notação para as gramáticas de linguagens de programação (TUCKER e NOONAN, 2010).

Em suma, a teoria de linguagens formais é fundamental para o entendimento da teoria de construção de compiladores. Este segundo tema é abordado em maiores detalhes na seção subsequente.

2.2.2 Teoria de construção de compiladores

Conjuntamente e paralelamente, a teoria de construção de compiladores surgiu com o objetivo de definir os parâmetros que permitem a construção de linguagens de programação e sua execução na máquina-alvo, por meio de tradutores ou compiladores. De maneira geral, um compilador é um programa que recebe como entrada um programa escrito em uma linguagem de programação – a linguagem fonte – e o traduz para um programa equivalente em outra linguagem – a linguagem alvo, preservando o significado sintático e semântico do programa nesse processo (GRUNE *et al.*, 2012).

Em suma, o processo de compilação é dividido em duas etapas principais, a de análise e a de síntese (também conhecidas como *front-end* e *back-end*), que podem ser subdivididas em diversas fases. Sumariamente, cada subetapa é responsável por mapear e transformar uma representação de um programa em outra (AHO *et al*., 2008).

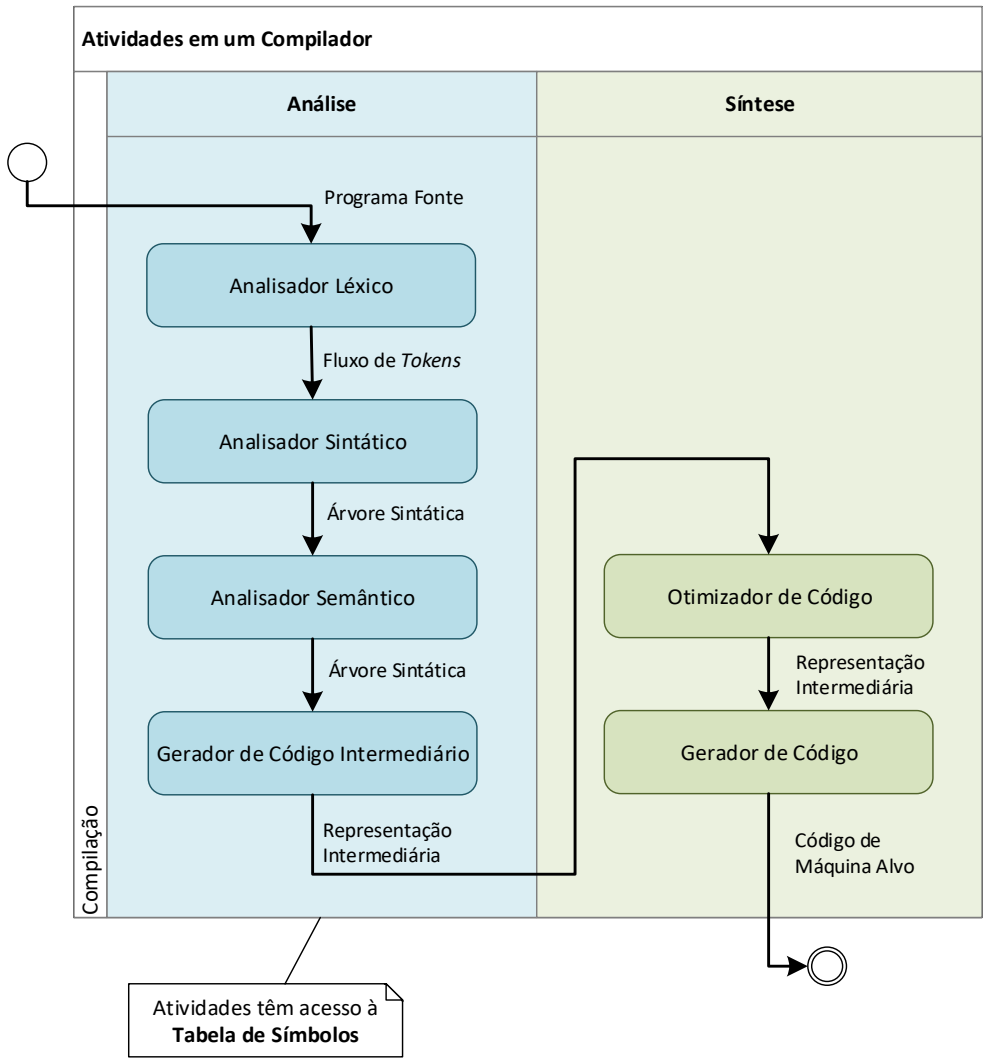
A etapa de análise, em geral, compreende as fases de análise léxica, sintática e semântica. A fase de análise léxica avalia se todos os símbolos presentes no código são válidos, extraindo tais elementos para a composição de uma estrutura de dados denominada de tabela de símbolos, a qual basicamente mapeia os construtos de um programa, tais como funções, constantes, variáveis e tipos de dados.

A fase de análise sintática, por sua vez, avalia se os símbolos estão encadeados corretamente de acordo com a especificação sintática da linguagem, criando representações abstratas deste programa nesse processo. Tais representações são denominadas de árvores abstratas sintáticas, as quais basicamente mapeiam a estrutura algorítmica de um programa.

A fase de análise semântica, ao seu turno, com base nas árvores sintáticas e na tabela de símbolos, avalia o programa semanticamente com o objetivo de garantir o uso correto dos elementos de um programa. A título de exemplo, ela avalia se as atribuições de variáveis, passagem de parâmetros e retorno de funções apresentam a tipagem correta.

Nesse âmbito, a Figura 6 apresenta as etapas de análise e síntese realizadas por um compilador.

Figura 6: Etapas e subetapa (fases) de um compilador



Fonte: Autoria própria via Diagrama de atividades em UML

Caso sejam encontrados erros, em qualquer uma das fases da etapa de análise, o compilador deveria apresentar mensagens esclarecedoras ao programador, de modo que ações corretivas possam ser efetuadas (AHO *et al.*, 2008). Ainda, durante a fase de análise é gerada uma representação intermediária do programa com base nas árvores sintáticas e tabela de símbolos, de modo a servir de entrada para a etapa de síntese.

A etapa de síntese, por sua vez, realiza a geração do código alvo, geralmente em uma linguagem de mais baixo nível, podendo utilizar as informações contidas na tabela de símbolos criada na etapa de análise. Ademais, na etapa de síntese, podem

existir variações de um compilador para outro, na qual etapas de otimização de código podem ser realizadas (AHO *et al*., 2008).

2.2.2.1 Análise Léxica

A primeira fase de um compilador tradicional, a análise léxica, tem como principal tarefa ler os caracteres de entrada do código-fonte de um programa, agrupá-los em lexemas e produzir como saída uma sequência de *tokens*, um para cada palavra ou símbolo especial encontrado durante a análise do código.

Durante esse processo, três elementos devem ser considerados: lexema, padrão e *token*. Segundo Aho *et al*. (2008) um *token* é um par, consistindo de um nome e um valor de atributo opcional (em caso de identificadores). O nome do *token* é um símbolo abstrato que representa um tipo de unidade léxica (*e.g*., palavra-chave, identificador, número etc.).

Por sua vez, um padrão de *token* pode ser descrito como a forma que os lexemas de um *token* podem tomar. No caso de uma palavra-chave da linguagem, o padrão é a sequência de caracteres que formam a palavra-chave (*e.g., class*). Já um lexema é a sequência de caracteres no código-fonte que associa um padrão a um *token* e pode ser considerada uma instância do *token* (AHO *et al.*, 2008). O Código 3 apresenta um trecho de código escrito em linguagem C no qual alguns lexemas são definidos.

Código 3: Exemplo de lexema utilizando a linguagem C

```
1 printf("total = %d\n", score);
```

Fonte: Autoria Própria

No Código 3 é possível identificar o lexema *printf* como sendo um padrão para o *token* PF definido na linguagem. O símbolo de parênteses "*(*" pode ser considerado um lexema para o padrão do *token* ABREPAR (*e.g*., Abertura de Parênteses). Seguindo tal linha de raciocínio, *"total = %d\n"* pode ser considerado um lexema para o padrão do *token* descrito como LITERAL. Por sua vez, *score* é o lexema para o padrão de *token* definido como ID (*i.e*., identificador de variável). Dessa forma, a Tabela 1 apresenta os *tokens* e lexemas que foram retirados do trecho de código-fonte apresentado no Código 3.

Tabela 1: Exemplos de *tokens* e lexemas

Token	**Lexema**
PF	printf
ABREPAR	(
LITERAL	"total = %d\n"
ID	score
FECHAPAR	)
VIRGULA	,
PONTOEVIRG	;

Fonte: Autoria Própria

Assim, ao se especificar uma linguagem, pode-se definir lexemas para palavras reservadas e operadores, porém para identificadores e números deve-se utilizar outra técnica, pois existem infinitas possibilidades de caracteres que constituem lexemas válidos para tal categoria de informação. Nesse caso, Expressões Regulares são utilizadas para definir formalmente um padrão. A Tabela 2 demonstra uma sintaxe de definição de expressões regulares, sendo *a* e *b* também expressões regulares.

Tabela 2: Exemplos de Expressões Regulares

(a)	Expressão *a*
ab	Expressão *a* seguida da expressão *b*
a | b	Expressão *a* ou expressão *b*
a*	Expressão *a* repetida um número qualquer de vezes
a+	Expressão *a* repetida pelo menos 1 vez
a?	Expressão *a* opcionalmente

Fonte: Autoria Própria

Utilizando o padrão da Tabela 2, é possível definir a expressão regular para compor o padrão de um identificador e um número. Tal exemplo é demonstrado no Código 4.

Código 4: Exemplos de expressões regulares na definição de identificadores

```
1  letra          ->  [A-Za-z]
2  digito         ->  [0-9]
3  identificador  ->  letra (letra | digito)*
4  numero         ->  digito*
```

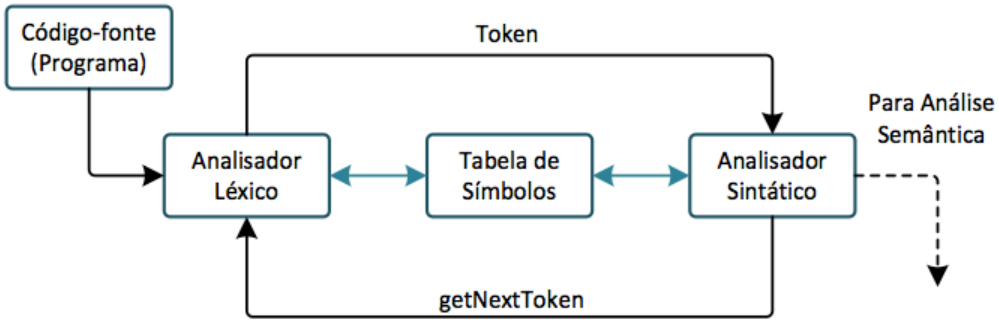
Fonte: Autoria Própria

Segundo Aho *et al.* (2008), as expressões regulares são uma importante notação para especificar os padrões de *tokens*, pois provêm uma forma concisa e flexível de identificar cadeias de caracteres que casem com padrões previamente estabelecidos na linguagem definida. Expressões regulares são baseadas em uma linguagem formal (*e.g.*, linguagens regulares) que pode ser interpretada por um processador de expressão regular que, neste caso, é o próprio analisador léxico.

Também é função do analisador léxico remover comentários e espaços em branco (*i.e.*, espaço, quebra de linha, tabulação e talvez outros caracteres que são usados para separar os lexemas na entrada), uma vez que este tem acesso ao código-fonte do programa. Ademais, se o programa fonte utilizar um pré-processador, a expansão de macro-comandos também pode ser realizada pelo analisador léxico (FERREIRA, 2015).

Em suma, além da identificação dos *tokens*, o analisador léxico também inicia a construção da tabela de símbolos com base nos identificadores encontrados no código-fonte (*e.g.,* variáveis, constantes etc.), incluindo estes na respectiva tabela. Na verdade, a fase de análise léxica é interdependente com a fase de análise sintática, acontecendo, normalmente, de forma iterativa e sequencial, conforme esboça a Figura 7.

Figura 7: Interação entre o analisador léxico e o analisador sintático



Fonte: Autoria Própria

Conforme esboçado na Figura 7, a interação é, normalmente, implementada fazendo-se com que o analisador sintático chame o analisador léxico. A chamada, sugerida pelo comando *getNextToken*, faz com que o analisador léxico leia caracteres

de sua entrada até que seja possível identificar o próximo *token* que será retornado ao analisador sintático (FERREIRA, 2015).

2.2.2.2 Análise Sintática

Conforme apresentado na Figura 7, o analisador sintático recebe do analisador léxico um conjunto de *tokens* que representam o programa fonte. O analisador sintático, portanto, verifica iterativamente se cada um dos *tokens* pertence à linguagem gerada pela gramática. A análise sintática tem por objetivo analisar se uma cadeia de *tokens*, extraída do código-fonte pertence a estrutura gramatical da linguagem definida, geralmente determinada por uma linguagem formal (AHO, 2008).

De acordo com Tucker e Noonan (2010), a sintaxe de uma linguagem de programação é uma descrição precisa de todos os seus programas gramaticalmente corretos. A sintaxe pode ser descrita por um conjunto de regras definidas em gramáticas livre de contexto. Atualmente, a BNF tem sido amplamente utilizada como linguagem para a construção de compiladores, conforme apresentado na Seção 2.2.1.

Ademais, o analisador sintático identifica cada *token* e categoriza apropriadamente cada qual, armazenando informações importantes sobre estes na tabela de símbolos. Em tempo, as tabelas de símbolos são basicamente estruturas de dados usadas pelos compiladores para conter informações sobre as construções do programa fonte. As informações são coletadas de modo incremental pelas fases de análise de um compilador, em especial pela análise sintática, e utilizadas pela etapa de síntese para gerar o código objeto. As entradas na tabela de símbolos podem conter informações como nomes de identificadores, seus tipos e escopo, assim como qualquer outra informação relevante ao contexto do programa. Nesse âmbito, o Código 5 e a Tabela 3 ilustram um exemplo particular de mapeamento de um programa, escrito em linguagem C, para uma tabela de símbolos.

Código 5: Programa exemplo em linguagem C

```
1  extern double getValor(int index);
2  double somador(int contador)
3  {
4      double soma = 0.0;
5      for (int i = 1;  i <= contador; i++)
6          soma += getValor(i);
7      return soma;
8  }
```

Fonte: Autoria Própria

Conforme apresenta o programa exemplo definido no Código 5, na linha 1 é basicamente declarada uma função externa que poderia acessar uma estrutura de dados qualquer, de modo a recuperar valores desta. Na sequência, na linha 2 é declarada outra função, esta responsável por somar todos os valores de uma determinada sequência de valores. Na prática, esse exemplo, em particular, apresenta vários símbolos, que apresentam tipos e escopos distintos, os quais são mapeados apropriadamente na Tabela 3.

Tabela 3: Exemplo de tabela de símbolos

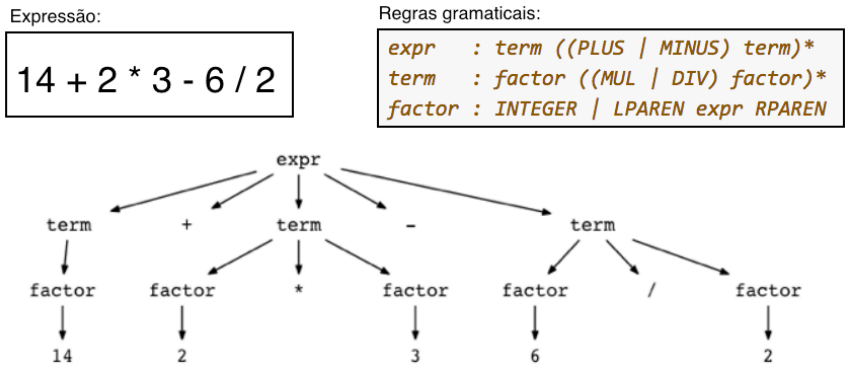
Símbolo	Tipo	Escopo
getValor	*function, double*	*extern*
index	*double*	*function param*
somador	*function, double*	*global*
contador	*int*	*function param*
soma	*double*	*block local*
i	*int*	*for-loop statement*

Fonte: Autoria Própria

Conforme ilustra o exemplo na Tabela 3, a tabela de símbolos mapeia os símbolos (*i.e.,* identificadores) de um programa a seus respectivos tipos, escopos. Também poderia mapear valores iniciais, entre outras informações pertinentes. Ademais, o compilador também poderia criar N tabelas de símbolos distintas por escopo. Nesse âmbito, uma classe em uma linguagem OO, por exemplo, teria sua própria tabela de símbolos, com uma entrada para cada atributo e método.

O processo de construção da tabela de símbolos está associado a construção de uma árvore de derivação (*parse tree*, do inglês), responsável por mapear a estrutura do programa em tal árvore. A título de exemplo, a Figura 8 ilustra um exemplo de expressão aritmética convertida em uma árvore de derivação.

Figura 8: Exemplo de árvore de derivação



Fonte: Adaptado de Spivak, 2015

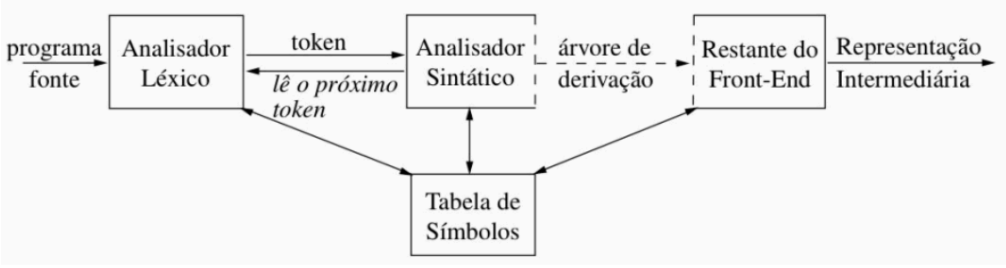
Basicamente, conforme ilustra a Figura 8, uma árvore de derivação é uma representação gráfica em que cada nó interno representa uma regra gramatical e cada nó folha representa um nó terminal (*e.g.,* 14, 2, +, *). Em suma, uma árvore de derivação representa todos os detalhes da sintaxe de um programa.

Em suma, a construção de uma árvore de derivação se dá por meio de algoritmos ou estratégias de processamento de gramáticas. Segundo Aho (2008) existem três estratégias gerais de análise sintática para o processamento de gramáticas: universal, descendente e ascendente. Os métodos de análise baseados na estratégia universal como o algoritmo Cocke-Younger-Kasami e o algoritmo de Earley podem analisar qualquer gramática. No entanto, esses métodos são muito ineficientes para serem usados em compiladores convencionais (AHO, 2008).

Nesse âmbito, os métodos utilizados em compiladores de uso geral são baseados nas estratégias descendente ou ascendente. Em suma, os métodos de análise descendentes constroem as árvores de derivação de cima (raiz) para baixo (folhas), enquanto os métodos ascendentes fazem a análise no sentido inverso, *i.e.,* começam nas folhas e avançam até a raiz construindo a árvore. Em ambas as estratégias, a entrada do analisador sintático é consumida da esquerda para a direita, um símbolo de cada vez.

Em suma, existem diversas tarefas que poderiam ser conduzidas durante a análise sintática, nas quais, destacam-se a coleta de informações sobre os *tokens* para posterior armazenamento na tabela de símbolos e criação da árvore de derivação. As demais atividades, como verificação de tipo e outros tipos de análise semântica, bem como a geração de código intermediário, se enquadram nas próximas etapas do processo de compilação (*i.e.,* seções 2.2.2.3 e 2.2.2.4), representados apenas como “Restante do *Front-End*”, conforme Figura 9.

Figura 9: Posição do analisador sintático no modelo de compilador



Fonte: Aho, 2008

Conforme apresenta a Figura 9, a tabela de símbolos é construída e utilizada em todas as fases do *front-end* de um compilador. É importante ressaltar que o analisador sintático geralmente está em melhor posição do que o analisador léxico para distinguir entre diferentes declarações de um identificador. Normalmente, na prática, o analisador léxico só retorna ao analisador sintático um *token* (*e.g.,* identificador), para que, só então, o analisador sintático decida se usará uma entrada na tabela de símbolos criada anteriormente ou se criará uma nova entrada para o identificador (AHO, 2008). Ademais, o analisador sintático é responsável por criar a árvore de derivação, a qual serve de entrada para a próxima fase (*i.e.,* análise semântica).

2.2.2.3 Análise Semântica

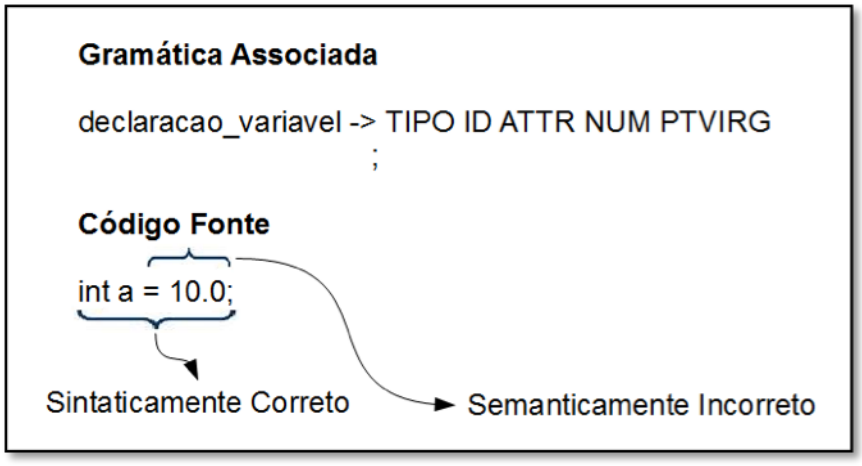
A fase de análise semântica, ao seu turno, utiliza a árvore de derivação, construída na etapa anterior, e as informações armazenadas na tabela de símbolos para verificar a consistência semântica do código-fonte de um programa com a definição da linguagem (AHO *et al*., 2008). Em linhas gerais, essa fase é responsável por garantir a consistência das regras semânticas em tempo de compilação, impondo algumas das seguintes condições (TUCKER e NOONAN, 2010):

- Que todos os identificadores referenciados no programa sejam declarados, garantindo, inclusive, a unicidade dos mesmos;
- Que os operandos para cada operador possuam um tipo apropriado;
- Que as operações de conversões envolvidas, por exemplo, inteiro para real, sejam inseridas onde for necessário.
- Outras validações pertinentes a linguagem a ser definida.

Além disso, por definição, a análise semântica tem por objetivo validar a significância do código-fonte do programa ou, até mesmo, seu comportamento em tempo de execução. A título de exemplo, no caso da definição de tipos de variáveis, considerando uma variável do tipo inteiro, na análise semântica é validada a atribuição de um valor inteiro a uma variável do mesmo tipo.

Na análise sintática, a validação está na estrutura gramatical, ou seja, se os símbolos estão encadeados corretamente de acordo com a especificação sintática da linguagem. A Figura 10 ilustra a diferença entre avaliação sintática e semântica.

Figura 10: Avaliação: análise sintática e análise semântica



Fonte: Ferreira, 2015

De fato, no exemplo ilustrado na Figura 10, os símbolos associados estão encadeados corretamente de acordo com a gramática definida. No entanto há uma inconsistência semântica na expressão, pois o literal de ponto flutuante não pode ser atribuído a variável de tipo inteiro. Por fim, é válido salientar que a análise semântica é uma etapa complementar a etapa de análise sintática, na qual validações pertinentes adicionais são realizadas no âmbito de garantir a consistência semântica de um programa.

2.2.2.4 Representação Intermediária

Após as devidas análises do programa fonte e respectiva construção da árvore de derivação e tabela de símbolos pertinentes, a próxima etapa do processo de compilação é a geração de uma Representação Intermediária (RI) apropriada. A RI pode ser representada na forma de uma estrutura de dados ou, mais comumente, na forma de código, normalmente baseado em uma linguagem intermediária (Aho *et al.,* 2008). Nesse âmbito, as representações em código podem ser:

- *Em alto nível*: A representação de código intermediário de alto nível está muito próxima da própria linguagem do código-fonte original. Ela pode ser

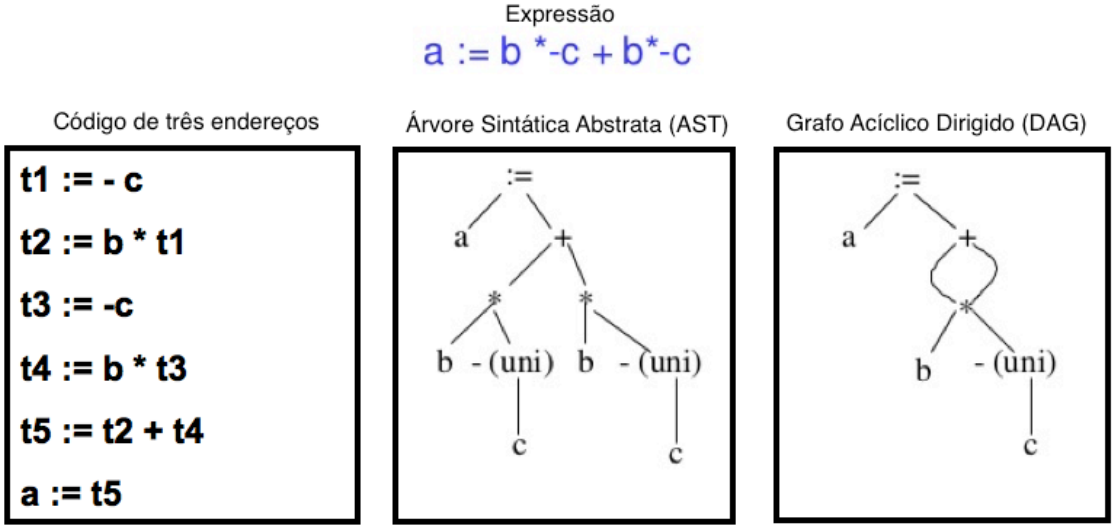
facilmente gerada a partir do código-fonte original e pode trabalhar com acesso a arrays, chamada de procedimentos etc. Entretanto, tende a ser mais orientada a otimizações genéricas, não especializadas na máquina alvo.

- *Em médio nível*: A representação de código intermediário de médio nível é próxima da máquina alvo, o que a torna adequada para alocação de registro e memória, seleção de conjunto de instruções etc., o que facilita otimizações dependentes da máquina.

O código intermediário pode ser definido em uma linguagem específica (*e.g., Bytecode* para Java) ou em uma linguagem independente (*e.g.,* código de três endereços). Na prática, o gerador de código intermediário recebe como entrada de sua fase predecessora (*i.e.,* análise sintática/semântica) uma árvore de derivação. Essa árvore de derivação pode então ser convertida em uma representação linear *e.g.,* notação pós-fixada ou código de três endereços. Basicamente, o código de três endereços é uma representação linearizada de uma árvore de derivação.

Por outro lado, o mesmo código pode ser apresentado em uma representação gráfica, por meio de árvores sintáticas abstratas (*Abstract Syntax Trees – AST*, do inglês) ou por meio de grafos acíclicos dirigidos (*Direct Acyclic Graph – DAG*, do inglês). Nesse âmbito, a Figura 11 apresenta um exemplo de tradução de uma expressão para a forma linear em código de três endereços e sua versão em uma *AST* e em um *DAG*.

Figura 11: Expressão representada em três representações intermediárias



Fonte: Aho *et al.*, 2008

Conforme apresenta a Figura 11, a expressão é apresentada no primeiro quadro na forma de código de três endereços, que basicamente segue a forma *SSA (Static Single Assignment*). A forma *SSA* determina, essencialmente, que cada registrador seja atribuído somente uma única vez (ROSEN *et al.,* 1988). No segundo quadro, por sua vez, é apresentada a representação desta mesma expressão em forma de uma *AST*. Por fim, no terceiro quadro, é apresenta na forma de um *DAG*, o qual basicamente simplifica a apresentação gráfica, eliminando estruturas redundantes (AHO *et al*., 2008).

Embora a geração de um código intermediário requeira um passo a mais na tradução, tornando o processo de compilação mais lento, o uso de uma RI independente de máquina, apresenta vantagens como a possibilidade de otimizações a nível de código intermediário, possibilitando que o código do alvo seja mais eficiente. Além disso, é possível fazer um reaproveitamento de código, facilitando a construção de geradores de código para diferentes plataformas de hardware, necessitando apenas que os módulos das últimas fases sejam refeitos, Outrossim, o uso de uma RI permite que sistemas de compilação como o GNU Compiler Collection (GNU, 2019) e o projeto LLVM (LLVM, 2019) suportem muitas linguagens de origem diferentes na geração de código alvo para muitas arquiteturas de destino diferentes. Essa abordagem cria uma camada intermediária no processo de compilação, por vezes denominada de *middle-end* (COOPER e TORCZON, 2011).

2.2.2.5 Otimização de código

De acordo com Grune *et al*. (2012), a fase de otimização é opcional, porém representa uma fase comum entre os compiladores tradicionais, antecedendo a geração de código final. Como o código gerado por meio da tradução orientada a sintaxe, diversas situações contendo sequências de código ineficiente podem ocorrer. Assim, a fase de otimização aplica um conjunto de heurísticas para detectar tais sequências e substituí-las por outras que removam as situações de ineficiência (GRUNE *et al*., 2012).

Na prática, a fase de otimização tem o propósito de melhorar o código da representação intermediária baseado na arquitetura da máquina-alvo. Melhorias comuns incluem encontrar expressões constantes e avaliá-las em tempo de compilação, reordenar código para melhorar o desempenho da cache, encontrar

subexpressões comuns e substituí-las por uma referência a um temporário e assim por diante. Um bom otimizador de código pode aumentar consideravelmente a velocidade de execução de um programa (TUCKER e NOONAN, 2010).

As técnicas de otimização que são usadas em compiladores precisam manter o significado do programa original e validar melhorias no código, porém dentro de limites razoáveis de esforço computacional. Segundo Grune *et al*., (2012), os compiladores usualmente permitem que o usuário especifique o grau de esforço desejado no processo de otimização. Por exemplo, no compilador *GCC*[5] pode-se utilizar o parâmetro de otimização em uma escala de 0 a 3, sendo -O0 (nenhuma otimização) e -O3 (máxima otimização, aumentando o tempo de compilação), bem como a opção -Os, que objetiva reduzir o espaço ocupado em memória.

Outras técnicas de otimização como a substituição de expressões que podem ser avaliadas durante o tempo de compilação pelos seus valores calculados, desmembramento de laços e substituição de operações (*e.g*., multiplicação por *shifts*) podem ser utilizadas no processo de compilação (AHO *et al*., 2008).

Isto posto, Aho *et al*. (2008) salienta que a geração de código objeto é uma tarefa árdua, principalmente devido às particularidades das máquinas-alvo. Assim, o código produzido pelo compilador deve aproveitar dos recursos especiais de cada máquina-alvo. Por fim, o código objeto pode ser uma sequência de instruções absolutas de máquina, uma sequência de instruções de máquina relocáveis, um programa em linguagem *Assembly* ou, até mesmo, um programa em outra linguagem, normalmente de mais baixo nível.

2.2.2.6 Geração de código

A fase final do compilador é a fase de geração de código, na qual é construído o código específico para o alvo designado. É importante salientar que o código-alvo pode ser tanto construído em uma linguagem de alto nível (*i.e.,* compilação de uma linguagem em alto nível para outra linguagem de alto nível, como por exemplo *Typescript* para *Javascript*), assim como de uma linguagem para código de máquina.

---

[5] O *GNU Compiler Collection* (*GCC*) é uma coleção de compiladores para linguagens de programação. Tal ferramenta permite compilar o código-fonte em binários executáveis para diversas plataformas. É distribuído pela *Free Software Foundation* (*FSF*) sob os termos da *GNU GPL.*

Neste último caso, a fase de geração de código é responsável por decidir quais instruções de máquina usar, como alocar registradores e outros detalhes dependentes da máquina (TUCKER e NOONAN, 2010).

A tradução do código-fonte de uma linguagem para outra está associada à tradução da representação da árvore gramatical, obtida por meio da busca de produções nas expressões do código-fonte, para a linguagem-alvo. Segundo Aho *et al*. (2008), embora tal atividade possa ser realizada para a árvore completa após a conclusão da análise sintática, em geral ela é efetivada por meio das ações semânticas associadas à aplicação das regras de reconhecimento do analisador sintático. Este procedimento é denominado tradução dirigida pela sintaxe.

Em geral, nos casos em que a compilação visa a construção de código para a máquina-alvo, a geração de código não se dá diretamente para a linguagem *Assembly* da máquina-alvo. Nesse caso, o analisador sintático gera código para uma máquina abstrata (intermediária), com uma linguagem próxima à da máquina, mas independente de processadores específicos. Adicionalmente, em uma nova etapa deste processo, tal código intermediário é traduzido para a linguagem *Assembly* desejada. Dessa forma, grande parte do compilador é reaproveitada para trabalhar com diferentes tipos de processadores (RICARTE, 2008).

2.2.3 Tecnologias para construção de compiladores

Em linhas gerais, os primeiros programas criados na histórica da computação foram escritos de forma manual até posteriormente serem construídos a partir de compiladores, conforme apresenta o breve resumo da história das linguagens no Apêndice A desta tese. Segundo Wexelblat (1981), os primeiros compiladores foram construídos em linguagem *Assembly*, traduzindo código de uma linguagem para código de máquina em uma única fase. Em contraste, a Seção 2.2.3.1 apresenta os compiladores de múltiplas fases, os quais foram dando origem a tecnologias que atualmente possibilitam a construção de sistemas de compilação completos, que integram todas as fases do processo de compilação, e principalmente permitem a compilação de múltiplas linguagens em um único sistema, com etapas comuns compartilhadas. A Seção 2.2.3.2, por sua vez, apresenta tecnologias de compilação para abordagens declarativas, especialmente para o PL. Por fim, a Seção 2.2.3.3

apresenta alternativas aos compiladores, especialmente no âmbito de interpretadores, bem como abordagens híbridas.

2.2.3.1 Sistemas de compilação para os paradigmas imperativos e funcionais

Em linhas gerais, um compilador poderia fazer parte de um sistema completo de compilação no qual, a partir de uma linguagem intermediária comum entre *front-ends* e *back-ends*, seria possível construir *N front-ends* compatíveis com *M back-ends*. Esta abordagem introduz um terceiro estágio intermediário chamado de *middle-end*, tornando possível combinar *front-ends* para diferentes linguagens com *back-ends* para diferentes CPUs, compartilhando as otimizações no estágio de *middle-end* (COOPER e TORCZON, 2011). Nesse âmbito, alguns sistemas de compilação utilizam essa abordagem para construir uma coleção de compiladores compatíveis entre si tais como, o GNU Compiler Collection (GCC) e o projeto LLVM.

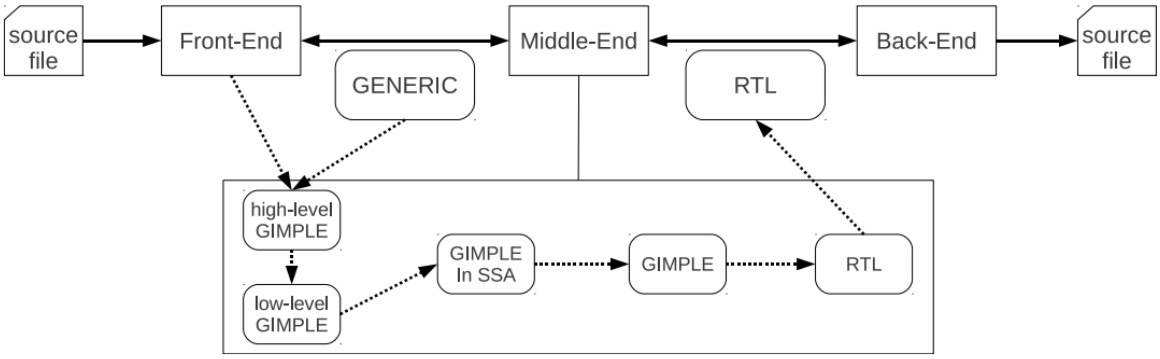
O GCC inclui *front-ends* para C, C ++, Objective-C, Fortran, Ada, Go e D, assim como bibliotecas para tais linguagens. Além dessas linguagens, existem diversos outros *front-ends* para diferentes linguagens que não foram integradas na distribuição oficial do GCC (GNU, 2019). Além disso, o GCC, por ser baseado em código livre, provê a possibilidade de que programadores criem novos *front-ends* para qualquer linguagem de programação, fornecendo para isso algumas opções de linguagens intermediária (GNU, 2019).

Inicialmente, o GCC era baseado na linguagem intermediária de baixo nível denominada de RTL (*Register Transfer Language*), na qual as instruções são descritas, praticamente uma por uma, em uma forma algébrica. Essa linguagem intermediária auxilia principalmente nas otimizações de baixo nível, porém apresenta limitações significativas que impedem que seja possível realizar otimizações de alto nível principalmente por se basear em tipos de dados limitados a palavras de máquina e não ter a capacidade de lidar com estruturas e matrizes. Nesse âmbito, o GCC passou por uma série de transformações, introduzindo novas linguagens intermediárias ao processo de compilação, com o objetivo de prover um conjunto completo de otimizações em nível da árvore, surgindo nesse ínterim a linguagem intermediária GIMPLE e seu superconjunto GENERIC (MERRILL, 2003).

Atualmente, o GCC utiliza as três linguagens intermediárias (*i.e.,* GENERIC, GIMPLE e RTL) para representar programas durante o processo de compilação.

Basicamente, a GENERIC é uma representação independente de linguagem, usada para servir como interface entre a fase de análise e a fase de otimização. Ademais, a GENERIC é capaz de representar programas escritos em todas linguagens suportadas pelo GCC. As demais linguagens (*i.e.,* GIMPLE e RTL), apesar de serem linguagens intermediárias, são usadas na otimização do processo de compilação (GNU, 2019), conforme ilustra a Figura 12.

Figura 12: Pipeline de compilação do GCC

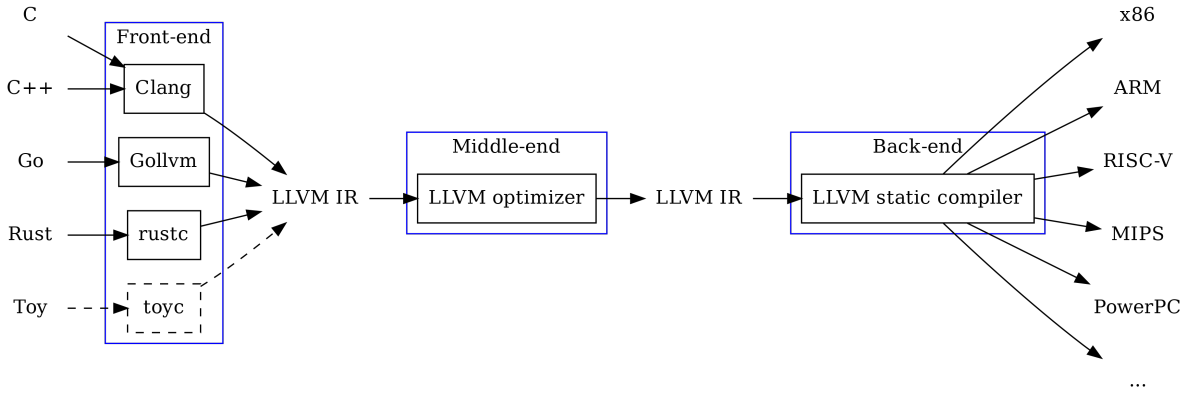


Fonte: Luo, 2014

O projeto LLVM, por sua vez, começou como um projeto de pesquisa na Universidade de Illinois, com o objetivo de fornecer uma estratégia moderna de compilação baseada em *SSA* capaz de suportar a compilação estática e dinâmica de linguagens de programação arbitrárias. Desde então, o LLVM tornou-se um projeto abrangente que consiste em vários subprojetos. O LLVM inclui *front-ends* para Ada, C, C++, D, Delphi, Fortran, Haskell, Julia, Objective-C, Rust e Swift, entre diversas outras linguagens (LLVM, 2019).

O núcleo do projeto LLVM é a representação intermediária (RI), uma linguagem de programação de baixo nível semelhante a *Assembly*, denominada de LLVM IR. Tal linguagem é fortemente tipada e baseada em RISC, acrônimo de *Reduced Instruction Set Computer* (em português, Computador com um conjunto reduzido de instruções) que basicamente abstrai os detalhes do programa alvo (LLVM, 2019). Segundo Eklind (2018), o processo de compilação no projeto LLVM é definido por três estágios, o *front-end*, *middle-end* e o *back-end*. Nesse âmbito, o código fonte é traduzido de um *front-end* para a LLVM IR, depois otimizado na fase de otimização, também considerada como de *middle-end*, para posterior tradução e compilação para código de máquina no *back-end*, conforme ilustra a Figura 13.

Figura 13: Pipeline de compilação do LLVM



Fonte: Eklind, 2018

Em suma, essa abordagem introduz um modelo de construção de compiladores cujas partes são reaproveitável e complementares, possibilitando a evolução incremental do sistema de compilação como um todo. Ademais, tal abordagem possibilita a construção de novas linguagens, a partir de seus *front-ends*, que aproveitariam todo uma base de compiladores, ou mais precisamente *back-ends* otimizados e preparados para geração de código de máquina para múltiplas arquiteturas, tornando toda a tarefa de construção de um novo compilador menos custosa.

É importante ressaltar que tais sistemas de compilação são normalmente utilizados na compilação de linguagens amparadas pelo paradigma imperativo e funcional, uma vez que normalmente mapeiam a estrutura algorítmica dos programas em forma de árvores e representações intermediárias outras, conforme o caso.

2.2.3.2 Tecnologias de compilação para abordagens declarativas

Conforme apresenta Paaki (1991), a construção de compiladores apesar de ser em sua maioria baseada na programação imperativa devido a fatores históricos, também pode ser implementada com linguagens de programação de outros paradigmas. Em seu trabalho, Paaki apresentou a construção da linguagem Edison em Prolog (paradigma lógico). Além disso, neste mesmo trabalho, Paaki apresentou a construção de uma versão do compilador em Pascal (abordagem procedural) e outra baseada nos sistemas PGS e GAG (abordagem declarativa).

Em geral, em termos de execução, o compilador construído em Prolog é ineficiente. Entretanto, conceitualmente, o Prolog fornece uma série de recursos de

alto nível para simplificar o processo de compilação, como a utilização da Gramática de Cláusulas Definidas (ou *Definite Clause Grammar – DCG*, em inglês) para análise, a qual representa um meio de expressar relações gramaticais, tanto para linguagens naturais quanto formais em lógica de primeira ordem. Tal técnica é uma alternativa a BNF, comum a implementações imperativas. Além disso, o Prolog também fornece um banco de dados interno para o gerenciamento de tabela de símbolos e a unificação com variáveis lógicas para a geração de código (PAAKI, 1991).

Outros trabalhos também vislumbraram a construção de ferramentas para a compilação tradicional em Prolog. Um dos primeiros a trabalhar nesta área foi Warren (1980), que em seu trabalho apresentou algumas técnicas para a construção de um compilador simples em Prolog. De maneira geral, em 1983, Warren projetou uma máquina abstrata para a execução do Prolog, consistindo de uma arquitetura de memória e um conjunto de instruções. Tal arquitetura foi denominada de *Warren Abstract Machine* (WAM) e tornou-se o alvo padrão para compiladores Prolog (AÏT-KACI, 1991; DIAZ e CODOGNET, 1995).

O principal objetivo de compilar o código Prolog para o código *WAM*, de mais baixo nível, é tornar a interpretação subsequente do programa Prolog mais eficiente. O código Prolog é razoavelmente fácil de traduzir para instruções *WAM* que podem ser interpretadas de forma mais eficiente. Além disso, as melhorias subsequentes de código e a compilação para código nativo são, muitas vezes, mais fáceis de executar na representação de mais baixo nível (WARREN, 1983).

Em Cohen e Hickey (1987), foi apresentado em detalhes as etapas de análise e síntese, por meio de tópicos que descrevem analisadores *bottom-up* e *top-down*, tradutores direcionados pela sintaxe, propriedades gramaticais, geração de código para *Assembly* e otimizações pontuais. Em outro trabalho, este desenvolvido por Russinoff (1992), foi estendida a teoria de Warren para fornecer um framework teórico com base matemática para o estudo da compilação em Prolog, demonstrando a equivalência semântica de dois interpretadores construídos nessa linguagem. Ainda neste trabalho foi apresentada uma versão de um compilador para a *WAM*.

Dedkov e Eadline (1995) apresentam um compilador capaz de traduzir código Prolog baseado em *WAM* para linguagem C, chamado de PTC (Prolog-To-C). Ademais, em comparação com emuladores de código WAM, a versão compilada se mostrou mais eficiente. Em outro trabalho, esse publicado por Codognet e Diaz (1995), foi apresentado o wamcc, outro compilador que traduz código Prolog para C,

apresentando como características a simplicidade, eficiência, portabilidade, extensibilidade e modularidade. É importante ressaltar, que a utilização da linguagem C como uma linguagem intermediária é uma prática comum, principalmente porque ela é portável para várias plataformas com código nativo e apresenta bons desempenhos de execução.

Diaz e Codognet (2001) também apresentaram outro compilador chamado de GNU Prolog, o qual foi reestruturado de maneira a ser mais otimizado que o próprio wamcc. Nesse âmbito, o GNU Prolog traduz o código fonte para uma linguagem intermediária independente de máquina, ainda que de baixo nível, chamada mini-assembly (especificamente projetada para o GNU Prolog). O arquivo resultante é então traduzido para o *Assembly* tradicional e posteriormente para a linguagem de montagem da máquina de destino. Isso permite que o GNU Prolog produza um executável autônomo nativo de um código-fonte escrito em Prolog. A principal vantagem deste esquema de compilação é produzir código nativo, com executáveis pequenos, que principalmente apresentam desempenhos superiores aos dos *solvers* comerciais tradicionais (GPROLOG, 2019).

Em relação a SBRs, normalmente tais soluções são interpretadas e baseadas em máquinas de inferência. Geralmente SBRs tradicionais apresentam linguagens proprietárias para definir as entidades utilizadas pelo motor de inferência. ILOG Rules, por outro lado, é um gerador de SBRs com a capacidade de traduzir módulos orientados a regras em módulos C++. Na prática, ILOG Rules utiliza o algoritmo Rete para a inferência no casamento ou *matching* dos fatos com as regras. O código fonte dos programas baseados no ILOG Rules compila para código C++, os quais são construídos de maneira completamente orientada a objetos. Com isso, tal ferramenta alia as vantagens de técnicas orientas a regras com as da orientação a objetos (ILOG, 1998).

2.2.3.3 Interpretadores e Abordagens Híbridas

Em contraste aos compiladores, os interpretadores não fazem nenhuma tradução de código. O interpretador atua como um sistema de simulação da máquina-alvo, que realiza o ciclo de execução com as instruções da linguagem definida em alto nível em vez das instruções da linguagem de máquina.

A interpretação pura tem a vantagem de permitir uma fácil implementação de operações de *debug* no nível de código-fonte, uma vez que todas as mensagens de erro, em tempo de execução, podem se referir as instruções exatas em suas respectivas linhas de código. Por exemplo, se um índice de matriz estiver fora de alcance, a mensagem de erro pode facilmente indicar a linha de origem e o nome da matriz.

Por outro lado, interpretadores apresentam desvantagens em termos de desempenho de execução, que tende a ser de 10 a 100 vezes mais lento que nos sistemas compilados. O principal motivo para a diferença em termos de execução é que a decodificação das declarações em linguagem de alto nível é muito mais complexa do que as instruções em linguagem de máquina, embora possam haver menos declarações do que instruções equivalentes em linguagem de máquina.

Ademais, independentemente de quantas vezes uma declaração é executada, ela deve ser decodificada sempre. Dessa forma, a decodificação da declaração, ao invés da conexão entre o processador e a memória, no caso da compilação, representa o maior gargalo da abordagem baseada em interpretação (SEBESTA, 2012).

Outra desvantagem da interpretação pura é que, muitas vezes, o programa alvo requer mais espaço em termos de memória. Além do programa fonte, a tabela de símbolos deve estar presente durante a interpretação. Além disso, o programa fonte pode ser armazenado em uma forma projetada para fácil acesso e modificação, em vez de estar em um formato que forneça um tamanho mínimo (SEBESTA, 2012).

Embora algumas linguagens de programação da década de 1960 (*e.g.,* APL, SNOBOL e LISP) foram implementadas puramente de forma interpretada, na década de 1980, a abordagem raramente era usada em linguagens de alto nível. No entanto, nos últimos anos, a interpretação pura retornou em algumas linguagens importantes como as de script para web, como o JavaScript e o PHP, que agora são amplamente utilizadas (SEBESTA, 2012).

Alguns sistemas apresentam uma abordagem intermediária entre compiladores e interpretadores puros, ou seja, traduzem programas de uma linguagem de alto nível para uma linguagem intermediária, projetada para permitir uma interpretação fácil. Esse método é mais eficiente do que a interpretação pura, uma vez que as declarações de linguagem original são decodificadas apenas uma

única vez. Tais implementações são chamadas de sistemas de implementação híbridos (SEBESTA, 2012).

A linguagem de programação Perl, baseada nos paradigmas funcional e imperativo, por exemplo, é implementada com um sistema híbrido. Os programas em Perl são parcialmente compilados para detectar erros antes da interpretação, além de promover uma simplificação no interpretador ao reduzir a necessidade de verificações de erro em tempo de execução (SEBESTA, 2012).

As primeiras implementações do Java também apresentaram abordagens híbridas. Sua forma intermediária, chamada de *byte code*, fornece portabilidade para qualquer máquina que tenha um interpretador de *byte code* e um sistema de tempo de execução associado. Tal mecanismo é popularmente chamado de Máquina Virtual Java. Atualmente, existem programas que traduzem o *byte code* Java para o código de máquina, visando uma execução mais otimizada em plataformas distintas.

Tais programas são chamados de *Just-in-Time* (*JIT*), os quais inicialmente traduzem programas para uma linguagem intermediária e durante a execução do programa, compilam o código intermediário em código da máquina à medida em que são chamados. Os sistemas *JIT* são amplamente utilizados para as versões mais atuais do Java. Além disso, as linguagens .NET também implementam um sistema JIT em sua essência (SEBESTA, 2012).

2.3 CONSIDERAÇÕES DO CAPÍTULO

Conforme apresentado ao longo deste capítulo, foi possível observar a influência dos paradigmas de programação para com as linguagens de programação criadas ao longo da história da computação.

Em geral, o paradigma mais dominante é o imperativo que apresenta algumas restrições em termos de modelo que implicam em acoplamentos implícitos e redundâncias de execução, tanto estruturais quanto temporais. Isso afeta diretamente a construção de programas, uma vez que se torna complexo desenvolver soluções que aproveitem efetivamente as capacidades de seus processadores, independente da linguagem utilizada.

A programação baseada nos princípios do Paradigma Imperativo (PI), de fato, frequentemente provoca acoplamentos em seus elementos de decisão (*e.g.*, aninhamentos de testes condicionais dentro de laços condicionais), tendendo a

produzir as redundâncias estruturais e redundâncias temporais salientadas. Isto efetivamente impacta negativamente no tempo de execução de programas e, principalmente, dificulta a modularização e, portanto, o reuso das partes do software, assim como também a paralelização ou distribuição do processamento.

O Paradigma Declarativo (PD) é uma alternativa ao PI que, apesar de facilitar em algo a programação por meio de suas linguagens de mais alto nível, normalmente usa algoritmos de inferência no casamento ou *matching* dos fatos com as regras. Tais algoritmos, porém, também apresentam em alguma medida redundâncias estruturais e temporais. Isso ocorre, principalmente, pelo modo como a inferência é realizada. Algumas soluções do PD, entretanto, reduzem a ocorrência de redundâncias de modo a melhorar a execução, tais como os Sistemas Baseados em Regras (SBR) apoiados no algoritmo Rete.

No entanto, tais soluções não resolvem completamente o problema e geralmente são compostas por estruturas de dados computacionalmente custosas. Ainda, estas soluções baseadas em inferência por busca em base de conhecimento passiva (compostas por fatos e regras), mantêm os elementos da base de fatos e os elementos da base de regras fortemente interdependentes, o que pode ser entendido como uma forma de acoplamento. Este tipo de solução orientada a busca impõem um processo de inferência monolítico. Isto dificulta a paralelização ou distribuição do processamento no PD.

Na verdade, tais paradigmas vigentes, PI e PD, bem como seus subparadigmas, poderiam ser classificados como sendo parte de um paradigma mais geral que poderia ser denominado de sequencial orientado a pesquisas. Isto se deve ao fato de que a computação, em geral, teve suas origens apoiadas na arquitetura Von Neumann e, portanto, são essencialmente sequenciais no seu âmago, sendo que as soluções de programação não raro se orientam por pesquisas em estruturas de dados e iterações orientadas a laços.

As linguagens de programação, a seu turno, são construídas com base em um ou mais (sub)paradigmas do PI ou PD, os quais norteiam a essência de concepção de programas, bem como a forma com que eles são interpretados ou executados na máquina alvo. Ao longo da história da computação, milhares de linguagens surgiram, principalmente amparadas pelos principais (sub)paradigmas de programação dominantes (*i.e.,* PP, POO, PL e PF englobados pelo PI ou PD) e muitas delas, inclusive, permitem um estilo de programação multiparadigmas.

Além disso, o capítulo apresentou a teoria geral de linguagens e compiladores, bem como os métodos e técnicas tradicionais adotados na construção de compiladores. Em suma, os compiladores construídos atualmente para as linguagens baseadas no paradigma geral que engloba PI e PD, aqui denominado de sequencial orientado a pesquisas, apoiam-se nessa teoria. Na prática, o processo de compilação é dividido em duas etapas (*i.e.,* análise e síntese), as quais podem ter variações em suas subetapas (fases), mas que geralmente se apoiam em representações intermediárias que mapeiam as estruturas algorítmicas e sequenciais em árvores.

Ao bem da verdade, a teoria de compilação apoiou-se naturalmente no paradigma geral sequencial oriundo do modelo computacional voltado principalmente para a arquitetura computacional de Von Neumann. Nesse âmbito, a teoria de compilação particularmente no tocante à representações intermediarias, naturalmente também se influenciou pela orientação a pesquisas.

Isto posto, é natural que novas linguagens de programação fiquem ainda atreladas ao paradigma geral de programação, que encontra seus enraizamentos na teoria de compiladores nos termos apresentados, bem como na própria arquitetura de computação. Apesar de existirem esforços para a criação e melhoria de linguagens de programação, tais linguagens ainda apresentam problemas como acoplamentos implícitos e redundâncias de execução simplesmente por serem baseadas nos paradigmas usuais.

Nesses termos todos dados, para os desenvolvedores utilizarem efetivamente os benefícios da computação paralelo-distribuída e, até mesmo, da computação tradicional, seria necessário a existência de novas soluções de programação. Nesse âmbito, o Paradigma Orientado a Notificações (PON) tem sido explorado como uma alternativa promissora para a construção de sistemas computacionais, inclusive atenuando erros de origem na própria computação sequencial particularmente no tocante a redundâncias e acoplamentos.

Basicamente, o PON apresenta um modelo computacional reativo, não orientado a pesquisa ou sequencialidade como nos paradigmas usuais. Em resumo, o modelo computacional do PON é baseado no relacionamento de entidades computacionais notificantes ditas ‘inteligentes’, comumente chamados de objetos-inteligentes ou ainda “agentes” (micro-agentes para ser mais preciso). Com base na colaboração entre tais entidades computacionais, o PON evita, por exemplo,

redundâncias em expressões lógico-causais e acoplamento nas unidades de software. Isto posto, o detalhamento do PON, com seu mecanismo de inferência colaborativa por notificações pontuais, é objeto de estudo dos próximos capítulos.

CAPÍTULO 3

PARADIGMA ORIENTADO A NOTIFICAÇÕES

Este capítulo, em especial, apresenta os conceitos teóricos fundamentais sobre o Paradigma Orientado a Notificações (PON). Além disso, apresenta os desdobramentos do paradigma tanto no estado da arte quanto no da técnica. Nesse âmbito, o capítulo está descrito e apresentado em seis seções. Primeiramente, a Seção 3.1 contextualiza os conceitos que fundamentaram o surgimento do PON. A Seção 3.2, por sua vez, apresenta a estrutura do PON e o funcionamento de seu modelo computacional. A Seção 3.3, particularmente, apresenta o PON em relação aos demais paradigmas. A Seção 3.4 apresenta os desdobramentos do PON tanto no âmbito do estado da arte quanto no da técnica. A seu turno, a Seção 3.5 apresenta as propriedades elementares do PON, vislumbradas essencialmente no estado da arte, mas não contempladas ainda por completo no estado da técnica. A Seção 3.6 apresenta os principais conceitos e funcionalidades do PON. Por fim, a Seção 3.7 apresenta algumas considerações sobre o PON.

A título de informação, caso o leitor apresente conhecimento sobre as seções deste capítulo, poderá não as ler ou postergar a leitura, sem prejuízo ao entendimento deste trabalho de doutoramento. É importante salientar, que algumas seções têm seus conteúdos baseados nos trabalhos do grupo de pesquisa do Paradigma Orientado a Notificações (PON), particularmente nos trabalhos de Banaszewski (2009), Ronszcka (2012), Linhares (2015), Santos (2017), Pordeus (2017) e Kerschbaumer (2018), que sintetizam os trabalhos do grupo em questão. Assim, este capítulo também serve como elemento de consulta para os temas do PON.

3.1 ORIGENS DO PON

Os conceitos que fundamentaram o surgimento do Paradigma Orientado a Notificações (PON) foram inicialmente propostos por Jean Marcelo Simão em sua dissertação de mestrado (SIMÃO, 2001) e em um artigo relacionado (SIMÃO e STADZISZ, 2002). Eles foram ainda mais aprofundados e detalhados em sua tese de doutorado (SIMÃO, 2005).

Nesses trabalhos, o autor propõe o mecanismo atualmente chamado de Controle Orientado a Notificações (CON) para sistemas de produção, mais precisamente, para sistemas de manufatura. Este mecanismo utiliza um metamodelo de controle discreto e holônico do CON de forma a aplicar instâncias deste sobre simulação realística de sistemas de manufatura em uma ferramenta chamada ANALYTICE II (SIMÃO, 2005). Isso permitiu simular realisticamente (quase emular por assim dizer) os chamados Sistemas Inteligentes de Manufatura, também chamados de Sistemas Holônicos de Manufatura (SIMÃO, 2001; SIMÃO e STADZISZ, 2002; SIMÃO, 2005; SIMÃO *et al*., 2006; 2008).

Nesse metamodelo do CON, as colaborações entre entidades "inteligentes" de manufatura são organizadas inclusive por meio de uma abordagem holônica (*i.e.,* sistêmica, integradora e sinérgica), na qual cada entidade de manufatura é integrada a um sistema computacional por meio de um recurso virtual e tem suas colaborações regidas por um sistema de controle "inteligente". Este metamodelo do CON trabalha via colaboração das entidades holônicas de manufatura com entidades holônicas de controle por meio de notificações pontuais (SIMÃO, 2001; SIMÃO e STADZISZ, 2002; SIMÃO, 2005; SIMÃO *et al*., 2006; 2008).

Na verdade, esse metamodelo de controle holônico atua sobre a ferramenta ANALYTICE II de forma a modifica-la para ser um simulador fidedigno de Sistemas de Manufatura Holônica. Este ferramental foi submetido a diversos testes e análises, via instância de controle holônico orientado a notificações, com os resultados documentados em diversos trabalhos (SIMÃO, 2001; SIMÃO e STADZISZ, 2002; SIMÃO, 2015, SIMÃO *et al*., 2006; SIMÃO *et al*., 2008).

Os sistemas discretos de controle e manufatura, holônicos ou não, não foram a única aplicação deste metamodelo do CON. Os autores perceberam, subsequentemente, que poderiam aplicá-lo em várias áreas, como: concepção de controle discreto em geral; inferências discretas em geral; e mesmo software de maneira genérica. Dessa maneira, a aplicação do metamodelo de notificações à construção de programas foi, à luz do tempo, denominada de Paradigma Orientado a Notificações (PON) (SIMÃO e STADZISZ, 2008).

O objetivo do PON, como paradigma de desenvolvimento, é tornar mais fácil a tarefa de criação e organização de projetos e artefatos no tocante a sistemas computacionais. Além da facilidade de composição de artefatos tais como código e mesmo circuitos, o PON busca tornar estes performantes, concorrentes e distribuíveis

(SIMÃO e STADZISZ, 2008; 2010; SIMÃO *et. al.,* 2010; 2012d; 2012e; LINHARES *et al*., 2014).

De maneira geral, o PON foi proposto como uma alternativa para o desenvolvimento de sistemas computacionais, visando diminuir problemas existentes nos atuais Paradigma Imperativo (PI) e Paradigma Declarativo (PD). Exemplos de problemas são as redundâncias de execução com o consequente mau uso de processamento e o correlato acoplamento excessivo entre entidades computacionais com a consequente dificuldade de reaproveitamento e de paralelização/distribuição das entidades computacionais de um sistema, conforme já detalhado no Capítulo 2 (SIMÃO e STADZISZ, 2008; BANASZEWSKI, 2009; LINHARES, 2015; PORDEUS, 2017; KERSCHBAUMER, 2018; OLIVEIRA *et al.,* 2018).

3.2 ESTRUTURA DO PON – MODELO COMPUTACIONAL

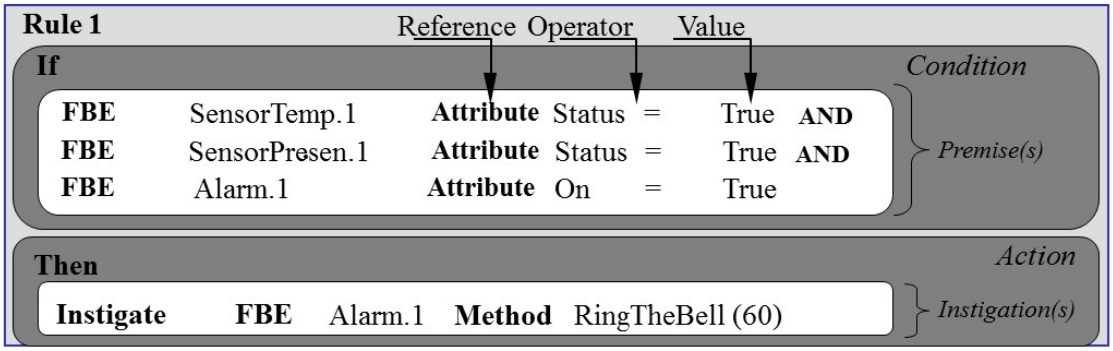
Em linhas gerais, o PON resolve certos problemas existentes nos paradigmas de programação citados. Na verdade, o PON unifica as principais características e as vantagens de PD (*e.g.,* representação do conhecimento em regras ou afins) e de PI (*e.g.,* flexibilidade de expressão e nível apropriado de abstração de módulos), resolvendo várias de suas deficiências e inconvenientes em aplicações monoprocessadas e multiprocessadas (SIMÃO e STADZISZ, 2008; 2009; BANASZEWSKI, 2009; SIMÃO *et al.,* 2012). Ademais, o PON foi inicialmente aplicado na composição de sistemas em software e depois expandido para a criação de circuitos em hardware digital.

O PON permite desacoplar expressões lógico-causais do código fonte ou modelo, quando considera cada uma destas expressões, e mesmo cada entidade factual correlata, como entidades “inteligentes” e notificantes, características essas que permitem que sistemas tenham desempenho e paralelismo/distribuição apropriados. Isto é diferente de programação ou desenvolvimento usuais, como os em OO e os em SBR, nos quais expressões causais são passivas e acopladas a outras partes do código, além de haver algum ou mesmo muito desperdício de processamento, conforme o caso (SIMÃO e STADZISZ, 2008; 2009; BANASZEWSKI, 2009; SIMÃO *et al.,* 2012).

No PON, a entidade “inteligente” que trata de uma expressão causal é chamada de *Rule*. As *Rules* gerenciam o conhecimento sobre qualquer

comportamento lógico-causal no sistema. O conhecimento lógico-causal de uma *Rule* provém normalmente de uma regra se-então, o que é uma maneira natural de expressão deste tipo de conhecimento. Nesse âmbito, a Figura 14 apresenta um exemplo de entidade *Rule*, justamente na forma de uma regra lógico-causal.

Figura 14: Exemplo de uma *Rule*

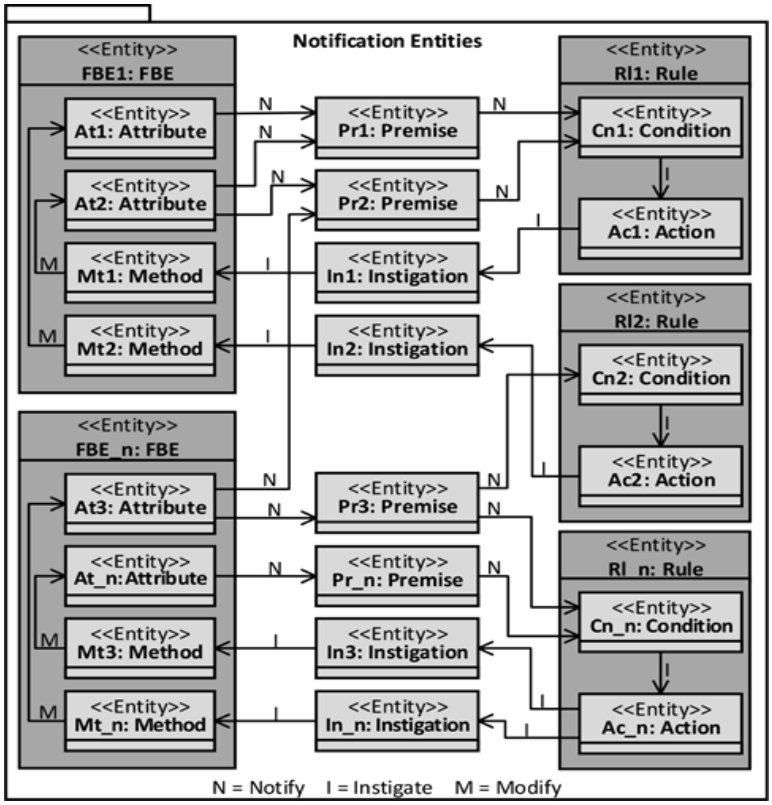


Fonte: Ronszcka *et al*., 2017b

Uma *Rule* tem uma entidade *Condition* e uma entidade *Action* como mostrado na figura em questão. A *Condition* trata da decisão da *Rule*, enquanto a *Action* trata da execução das ações da *Rule*. Assim, *Condition* e *Action* trabalham para realizar o conhecimento causal da *Rule*. Na verdade, tanto a *Condition* quanto a *Action* são entidades "inteligentes" agregadas nela, conforme modelado em Linguagem de Modelagem de Sistemas (do inglês, *System Modeling Language* - *SysML*) na Figura 15 (SIMÃO e STADZISZ, 2008; 2009; BANASZEWSKI, 2009; SIMÃO *et al.,* 2012).

Esta *Rule* apresentada faria parte de um sistema de alarmes por correlação de sensores, na qual sensores são tratados a partir de entidades ou objetos "inteligentes", sendo entidades factuais nesse contexto dado ou fato-execucionais, mais precisamente. A *Condition* dessa *Rule* em questão lida com a decisão de disparo de alarme a partir de um '*SensorTemp.1*' (Sensor de Temperatura), de um '*SensorPresen.1*' (Sensor de Presença) e de um 'Alarm.1' (Alarme). Na verdade, cada um destes objetos "inteligentes", analisáveis por *Conditions*, é chamado de *Fact Base Element* (*FBE*) no PON (SIMÃO e STADZISZ, 2008; 2009; BANASZEWSKI, 2009; SIMÃO *et al.,* 2012).

Figura 15: Entidades Constituintes do PON



Fonte: Kerschbaumer, 2018 via Diagrama de Blocos em SysML

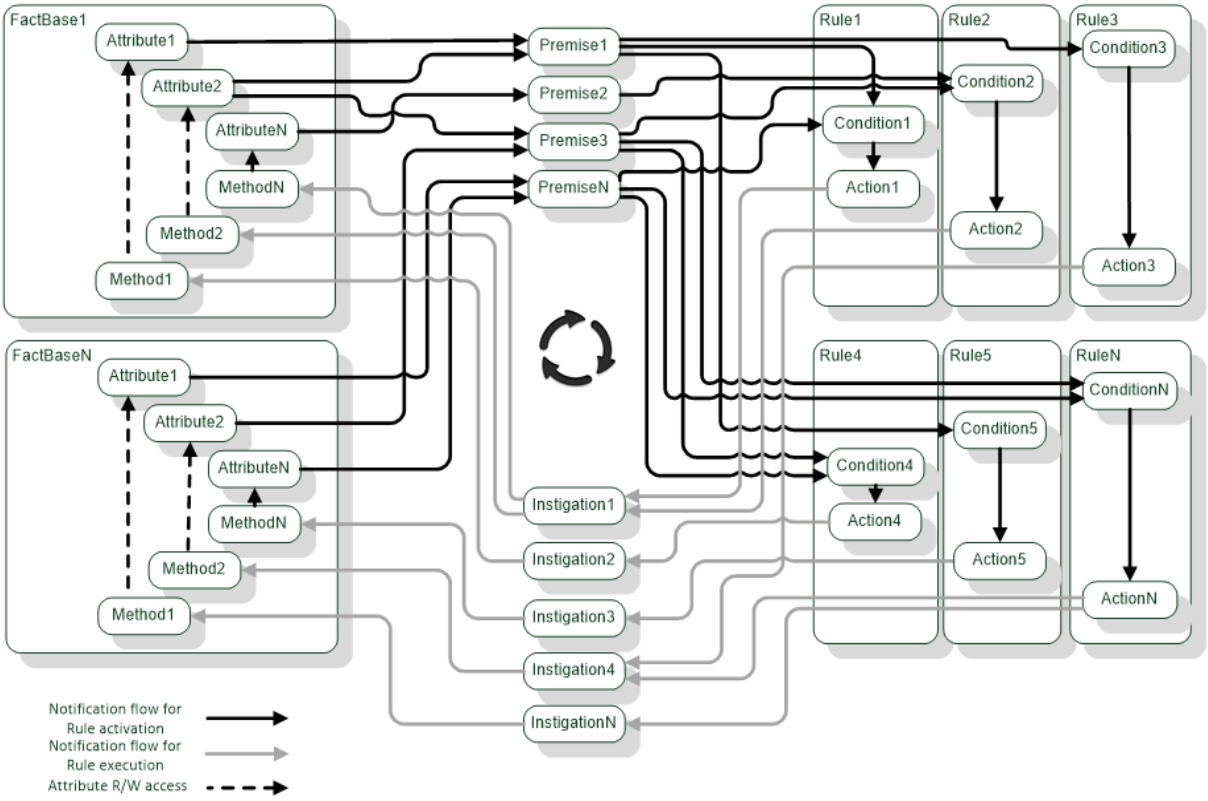
Como ilustrado na Figura 14, a *Condition* daquela *Rule*, em especial, é composta por três *Premises* que se constituem em outro tipo de entidade ou objeto "inteligente". Essas *Premises* em questão fazem as seguintes verificações sobre as *FBEs*: a) *o Sensor de Temperatura 1 está acionado*? b) *o Sensor de Presença 1 está acionado*? c) *o Alarme 1 está ativo*? Assim, conclui-se, em geral, que os estados dos atributos das *FBE* compõem os fatos a serem avaliados pelas *Premises* (SIMÃO e STADZISZ, 2008; 2009; BANASZEWSKI, 2009; SIMÃO *et al.,* 2012).

Na verdade, os estados de cada um dos atributos de uma *FBE* são tratados por meio de uma entidade ou objeto "inteligente" chamado *Attribute*. Além do mais e principalmente, para cada mudança de estado de um *Attribute* da *FBE*, ocorrem automaticamente avaliações somente e precisamente nas *Premises* relacionadas, com eventuais mudanças nos seus estados. Igualmente, a partir da mudança de estado em *Premises*, ocorrem automaticamente avaliações somente e precisamente nas *Conditions* relacionadas com eventuais mudanças de seus estados (SIMÃO e STADZISZ, 2008; 2009; BANASZEWSKI, 2009; SIMÃO *et al.,* 2012).

Isto tudo se dá por meio de uma elegante cadeia de notificações entre entidades ou objetos "inteligentes", o que se constitui no ponto central da inovação do

PON. Esta elegante, desacoplante e precisa cadeia de inferência por notificações é esboçada na Figura 16 (SIMÃO e STADZISZ, 2008; 2009; BANASZEWSKI, 2009; SIMÃO *et al.,* 2012).

Figura 16: Representação do Ciclo de Inferência por Notificações do PON



Fonte: Linhares, 2015

Em suma, cada *Attribute* notifica pontualmente as *Premises* relevantes sobre seus estados (principalmente e normalmente na mudança de estado) somente quando se fizer efetivamente necessário. Cada *Premise*, por sua vez, notifica as *Conditions* relevantes dos seus estados (principalmente e normalmente na mudança de estado) usando o mesmo princípio (SIMÃO e STADZISZ, 2008; 2009; BANASZEWSKI, 2009; SIMÃO *et al.,* 2012).

Baseado nestes estados notificados é que a *Condition* pode ser aprovada ou não, mantendo registro dos estados notificados e os relacionando por algum operador lógico como E, OU, OU Exclusivo ou, até mesmo, combinações deles. Na verdade, a *Condition* pode ser efetivamente aprovada se, além de estar em estado verdadeiro, tiver eventuais conflitos e eventuais questões de determinismo tratadas, sendo que há inclusive soluções baseadas em contra-notificações para tal (SIMÃO e STADZISZ, 2010). Uma vez que uma dada *Condition* seja efetivamente aprovada, a respectiva

Rule é normalmente ativada executando sua *Action* (SIMÃO e STADZISZ, 2008; 2009; BANASZEWSKI, 2009; SIMÃO *et al.,* 2012).

Por sua vez, uma *Action* também é uma entidade ou objeto "inteligente" que se conecta a objetos "inteligentes" de outro tipo, as *Instigations*. No exemplo considerado de sistema de alarmes por correlação de sensores, a *Action* contém uma *Instigation* para: a) *ativar o Alarme 1 por 60 segundos*. Efetivamente, o que uma *Instigation* faz é instigar um ou mais métodos responsáveis por realizar serviços ou habilidades de uma *FBE*, sendo assim mais um elemento organizacional e desacoplante (SIMÃO e STADZISZ, 2008; 2009; BANASZEWSKI, 2009; SIMÃO *et al.,* 2012).

Ademais, cada método de uma *FBE* é também tratado por uma entidade ou objeto "inteligente", que é chamado de *Method*. Geralmente, a execução de um *Method* muda o estado de um ou mais *Attributes*. Na verdade, os conceitos de *Attribute* e *Method* representam uma evolução dos conceitos de atributos e métodos de classe/objeto da OO. Exemplos das diferenças evolutivas são o desacoplamento implícito da entidade proprietária e a "inteligência" colaborativa pontual para com *Premises* e *Instigations* (SIMÃO e STADZISZ, 2008; 2009; BANASZEWSKI, 2009; SIMÃO *et al.,* 2012).

Em tempo, a criação das conexões entre as entidades notificantes[6] se dá em tempo de construção em ambiente ou linguagem de desenvolvimento apropriada. Quando, por exemplo, em tempo de construção uma *Premise* referencia um *Attribute*, este cadastra a *Premise* em questão para ser notificada (isto em tempo de execução) sobre mudança de estado pertinente (SIMÃO e STADZISZ, 2008; 2009; BANASZEWSKI, 2009; SIMÃO *et al.,* 2012).

Isto considerado, esta seção apresentou o PON de maneira efetivamente resumida, sendo que os detalhamentos estão especificados nas dissertações, teses, patentes, artigos e demais publicações pertinentes. Outrossim, as próximas subseções apresentam de forma detalhada os desdobramentos do PON tanto no

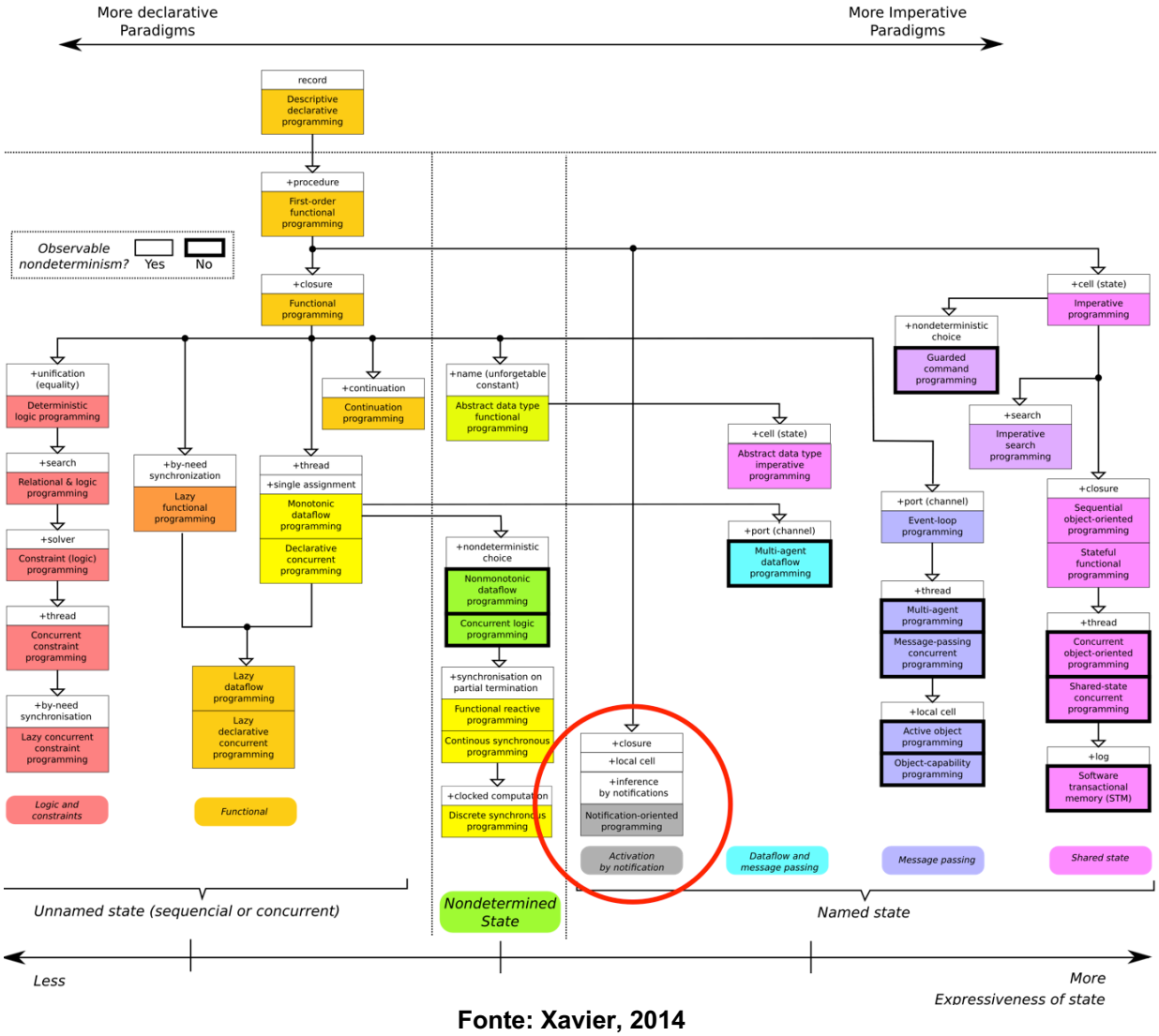
[6] De acordo com Simão *et al*. (2012a), a solução de inferência do PON, responsável por determinar o fluxo de execução de uma aplicação desenvolvida sob os princípios desse, não se trata apenas de uma aplicação do conhecido padrão de projeto *Observer*. Tal solução é dita como uma extrapolação desse padrão, aplicada no âmbito do cálculo lógico-causal desse paradigma, por meio de notificações pontuais a pequenas entidades reativas em um arranjo inovador (RONSZCKA, 2012; RONSZCKA et al., 2017a).

estado da arte quanto no da técnica, referenciando apropriadamente cada um dos trabalhos relacionados.

3.3 PON EM RELAÇÃO AOS DEMAIS PARADIGMAS

Com base nas principais características do PON, Xavier (2014) buscou encaixar o PON na taxonomia proposta inicialmente por Van Roy (2009), relembrando a Figura 4, apresentada anteriormente no Capítulo 2, a qual apresenta a taxonomia dos principais paradigmas de programação usuais. Tal adaptação é apresentada na Figura 17.

Figura 17: Taxonomia de Paradigmas de Programação incluindo o PON



Fonte: Xavier, 2014

Conforme apresenta a Figura 17, o grafo apresenta a taxonomia completa, encaixando o PON de acordo com suas características. Primeiramente, a escolha do posicionamento do paradigma (graficamente e semanticamente) se deve pela estrutura intrínseca da própria taxonomia: expressividade de estado (ativação por notificações tem expressividade menor que passagem de mensagem e maior que estado não nomeado), forma de programação (mais declarativa ou mais imperativa – eixo horizontal) e soma de características únicas (eixo vertical). Em segundo lugar, a grosso modo, os paradigmas adjacentes são os parentes próximos (ou menos distantes) do paradigma. Por último, é o local mais equidistante entre os paradigmas "ancestrais" do PON: Orientação a Objetos e Declarativo (XAVIER, 2014). Ademais, segundo Xavier (2014), o PON apresenta as principais características, segundo os principais quesitos propostos na taxonomia de Van Roy (2009):

a) ***record***: *FBEs* com propriedades (*Attributes*), contém uma tupla <nome-valor>, os quais podem ser construídos, associados, reorganizados, recompostos e reposicionados dinamicamente;
b) ***closure***: cada unidade computacional é uma *closure* em PON, desde o nível 'micro' (*Attribute*, *Premise*, *Instigation*, *Method*) até 'macro' (*FBE*, *Condition, Action* e *Rule*);
c) ***concorrência***: distribuição das próprias entidades (unidades computacionais), por conta do modelo de execução. A distribuição, neste caso, não fica essencialmente a cargo da programação;
d) ***expressividade***: está presente em cada unidade computacional do PON (*local cell*), que gerencia o seu próprio estado e suas próprias responsabilidades;
e) ***não-determinismo observável***: condição alcançável por meio de técnicas apropriadas de resolução de conflitos, as quais podem mitigar a ocorrência de não-determinismo. Não obstante, Simão e Stadzisz (2010) propõe uma técnica baseada em contra-notificações para prover determinismo no PON;
f) ***características singulares***: célula local (estado) e inferência por notificações (*inference by notifications*).

3.4 DESDOBRAMENTOS DO PON NO ESTADO DA ARTE E DA TÉCNICA

Em linhas gerais, o PON foi inicialmente materializado na forma de arquétipos/*frameworks*. O PON também alcançou o âmbito da modelagem de software, trazendo novas técnicas mais apropriadas para si. Isto tudo proporcionou ao PON ser aplicado para conceber e executar software prototipais em uma série de contextos como sistemas de informação, jogos, sistemas embarcados, sistemas ubíquos e outros, alcançando resultados promissores em tempo de desempenho devido às renuncias a redundâncias e ao desacoplamento implícito proporcionado pela Inferência Orientada a Notificações (ION) (BANASZEWSKI, 2009; VALENÇA, 2012; RONSZCKA, 2012; FERREIRA, 2015).

Ao bem da verdade, inclusive em função do desacoplamento implícito, o PON também vem sendo testado em domínio de multiprocessamento tanto para multicore quanto para lógica reconfigurável – *FPGA* (*Field-Programmable Gate Array*). Neste sentido, o PON transpassou o domínio do software sendo também uma solução para desenvolvimento de hardware. Na verdade, ele tornou-se um paradigma de computação, além de um paradigma de desenvolvimento de software e hardware específico, principalmente à medida que foi desenvolvida uma nova arquitetura computacional para ele chamada de ARQPON. A ARQPON é diferente da tradicional arquitetura von Neumann e similares, havendo prototipação dela em *FPGA* e respectivo simulador em software (WITT *et al*., 2011; PETERS, 2012; LINHARES, 2015; PORDEUS, 2017; KERSCHBAUMER, 2018).

Ainda, o PON está também sendo generalizado para outros domínios como lógica nebulosa/*fuzzy*, redes neurais, ontologias e sistemas multiagentes sendo que alguns trabalhos estão mais avançados que os outros (MELO *et al.,* 2015; MELO, 2016; SCHÜTZ *et al.,* 2018; SCHÜTZ, 2019). Inicialmente, isto tem se dado no PON enquanto paradigma de software, mas já tendendo igualmente ao hardware. Essas evoluções do PON, como a do PON de software para o hardware alcançando a computação em si e a generalização para diversas áreas de aplicabilidade, também é percebida por novos estudos que expandem sua aplicabilidade em engenharia de requisitos, de software, mesmo de sistemas e, quiçá, de testes (WIECHETECK, 2011; KOSSOSKI, 2015; MENDONÇA *et al*., 2015; MENDONÇA, 2016).

Nesse âmbito, de modo a apresentar em maiores detalhes os desdobramentos do PON, ao longo dos últimos anos, primeiramente, a Seção 3.4.1

apresenta as materializações do PON em Software. A Seção 3.4.2, particularmente, apresenta as materializações do PON em Hardware. A seu turno, a Seção 3.4.3, mostra os avanços relacionados a Engenharia de software para o PON. Por fim, a Seção 3.4.4, apresenta as iniciativas em Inteligência Computacional/Artificial vislumbradas a partir do modelo computacional do PON.

3.4.1 Materializações em Software

A primeira implementação dos conceitos do PON foi realizada sobre um arquétipo ou *Framework* desenvolvido em linguagem de programação C++. O objetivo deste *Framework* é oferecer uma interface de programação (API - *Application Programming Interface*) para o desenvolvimento de aplicações seguindo o modelo do PON, com abstrações necessárias para compor as *FBEs* e *Rules* (SIMÃO *et al*., 2012a; SIMÃO *et al*., 2017a).

O *Framework* PON C++ prototipal (ou referencial) foi proposto por Simão em 2007 (SIMÃO e STADZISZ, 2008; 2009; SIMÃO *et al.*, 2012a; SIMÃO *et al*., 2017a), remontando ao *Framework* que implementa o metamodelo do Controle Orientado a Notificações (CON) sobre a ferramenta de simulação de sistemas de manufatura ANALYTICE II (SIMÃO, 2005), a qual foi desenvolvida pelo próprio grupo de pesquisa naquela época. Este *Framework* foi refeito no trabalho de Banaszewski (2009) gerando o chamado *Framework* PON C++ 1.0 (ou original) (SIMÃO *et al*., 2017b). Subsequentemente, uma nova versão do *Framework,* otimizada e orientada a padrões de projeto, foi elaborada nos trabalhos de Valença (2012) e Ronszcka (2012) sendo chamada de *Framework* PON C++ 2.0 (ou otimizado) (RONSZCKA *et al*., 2017a; SIMÃO *et al.*, 2017c). Este ainda se constituía, até então, na principal materialização estável do PON para software (VALENÇA, 2012; RONSZCKA, 2012; RONSZCKA *et al*., 2017a).

De fato, as implementações em forma de *frameworks* são implementações compostas por um conjunto de classes cujos métodos são materializados em PI/POO, sendo que sua implementação se baseia em percorrer estruturas de dados (originalmente fornecidas pela *STL– Standard Template Library* na versão 1.0 do *Framework* – e subsequentemente por biblioteca própria na versão 2.0 do *Framework*) para avaliação das relações lógico-causais e fazer o envio de notificações na forma de chamada de métodos. Tal abordagem é desvantajosa à filosofia do PON, pois há

uma enorme dependência de estruturas de dados e a execução sobre tais estruturas se dá de forma sequencial, decorrendo assim em degradação de desempenho (BANASZEWSKI, 2009; VALENÇA, 2012; RONSZCKA, 2012).

Neste âmbito, Valença (2012) efetuou uma série de otimizações com objetivo de aprimorar o desempenho de execução das aplicações sobre *Framework* PON C++. O resultado deste trabalho foi a versão 2.0 do *Framework*, a qual é baseada em uma variedade de estruturas de dados mais otimizadas do que as equivalentes fornecidas pela *STL*, por exemplo, vetores (PONVECTOR), listas (PONLIST) e tabelas *hash* (PONHASH). Tais otimizações fizeram o *Framework* PON C++ 2.0 apresentar ganhos de desempenho em diversas aplicações quando comparado à sua versão original (VALENÇA, 2012; RONSZCKA *et al*., 2017a).

Ademais, segundo Ronszcka (2012), o *Framework* PON C++ 2.0 permite uma rápida prototipação de aplicações PON para teste sobre plataformas de computação convencionais. Entretanto, mesmo após otimizações feitas no âmbito do *Framework,* experimentos foram conduzidos em (SIMÃO *et al*, 2012b; 2012c) sendo que os resultados ainda não se mostraram satisfatórios em termos de desempenho, conforme o cálculo assintótico do PON (SIMÃO, 2005, BANASZEWSKI, 2009). Tal fato se deve principalmente ao uso de estruturas de dados dispendiosas em termos de desempenho para o processo de inferência.

Ainda, no âmbito de software, é importante salientar que existem outras materializações prototipais. O *Framework* PON C++ prototipal foi adaptado por Weber para trabalhar com *multithreads* (WEBER *et al*., 2010). Subsequentemente, Belmonte e Ronszcka adaptaram o *Framework* PON C++ 2.0 para trabalhar com *Threads*, o que resultou no chamado *Framework* PON C++ 3.0, aplicando-o para aplicações *multicore* a fim de demonstrar a capacidade multiprocessada do PON já obtendo resultados positivos (BELMONTE *et al*., 2016).

Além das adaptações dos *Frameworks* em C++, o *framework* também apresenta versões desenvolvidas em linguagem Java e C# (HENZEN, 2015). O *Framework* PON Java foi uma adaptação do Framework PON C++ 1.0 sendo que, não obstante, em resultados preliminares o desempenho de execução dele se assemelharam aos do *Framework* PON C++ 2.0 (HENZEN, 2015). O primeiro *Framework* PON C# segue o mesmo quadro e resultados do *Framework* Java. Entretanto, além dos resultados serem preliminares não houve comparações para com aplicações semelhantes em POO/PI (HENZEN, 2015). Em Mendonça, há o relato

de uma aplicação PON feita neste *Framework* PON Java mas sem comparações com POO/PI (MENDONÇA, 2016). Ainda, foi desenvolvido uma evolução do *Framework* PON C#, o qual foi usado em aplicação híbrida com POO para simulação de um ambiente voltado para Internet das Coisas (*IoT – Internet of Things*, em inglês), mais especificamente para ambientes senscientes (OLIVEIRA, 2016; OLIVEIRA *et al*., 2018).

3.4.2 Materializações em Hardware

Com o intuito de construir sistemas concebidos seguindo o modelo do PON de maneira mais fidedigna ao paralelismo, vislumbrou-se a materialização do PON em hardware. Assim técnicas, modelos de circuitos e estruturas arquiteturais de hardware foram propostas, permitindo a construção de ambientes nos quais o mecanismo de notificações possa executar, de fato, em paralelo. Tais proposta também objetivam minimizar questões de *overhead* de processamento presente nas soluções propostas em software.

A primeira proposta de materialização de uma aplicação PON em *hardware*, foi feita por Witt *et al*. (2011), no qual foi proposto um conjunto de circuitos sequenciais/combinacionais que implementam as entidades PON em hardware (PORDEUS *et al*., 2016; SIMÃO *et al*., 2012b; WITT *et al*., 2011). Tal modelo é conhecido como PON-HD prototipal, sendo que tais circuitos podem ter suas interconexões sintetizadas em *FPGAs*, de forma a implementar a cadeia de notificações do PON para cada aplicação em específico. Dado a pequena complexidade do funcionamento de cada uma das entidades do PON, estas podem ser modeladas em hardware como blocos lógicos (AND, OR, NOT) e blocos sequenciais (*latches* e *flip-flops*) (WITT *et al*., 2011)

Em um primeiro experimento, Witt *et al.* (2011) propuseram a comparação de duas versões de um simulador de sistema de telefonia, ambos implementados em hardware. Neste contexto de experimentos, uma das versões foi criada com base nos princípios do POO e a outra com base nos princípios do PON de forma tal a comparar tais implementações em ambiente de hardware digital, sem influências outras, como camadas de software tais quais as do SO (Sistema Operacional) e de *APIs*.

Em tempo, a implementação foi baseada no trabalho de Linhares *et al.* (2011). Como resultado da comparação dos simuladores, quanto aos resultados qualitativos,

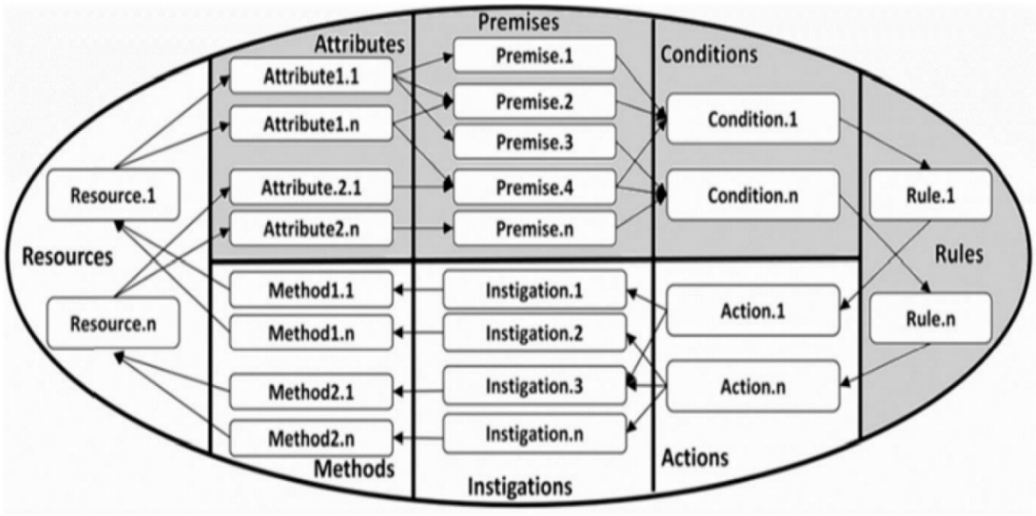
a solução PON foi mais intuitiva e fácil de ser composta em hardware do que a solução baseada em POO. Outrossim, em termos de desempenho, a versão em PON apresentou melhores resultados em relação a versão em POO (WITT *et al*., 2011). Em suma, tais resultados se deram pelas características de desacoplamento das entidades do PON, que permitem uma execução paralela, e desta forma, a implementação em *hardware* traz potenciais ganhos de desempenho em relação a implementações convencionais do PON realizadas puramente em software (WITT *et al*., 2011).

Em tempo, Jasinski (2012) propôs uma solução que permite fazer a descrição de uma aplicação PON em alto nível e transformá-la (por meio de mapeamento) em código do *Framework* PON VHDL Prototipal, tornando assim possível a execução direta do PON-HD. Os elementos PON suportados pela solução em questão (nomeadamente *Attributtes*, *Premises*, *Conditions* e *Instigations*) são descritos através da linguagem YAML (*YAML Ain't Markup Language*). Após a transformação, são gerados circuitos sequenciais e combinacionais que implementam a aplicação baseados nos elementos do *Framework* PON VHDL Prototipal (JASINSKI, 2012; LINHARES, 2015). Entretanto, a solução PON-YAML apresenta algumas restrições quanto à implementação dos conceitos do PON. Tais restrições referem-se à resolução de conflitos entre *Rules*, à definição de prioridades entre *Rules*, ao suporte a *Methods* desenvolvidos segundo o PI, à execução de *Instigations* de maneira sequencial (executam somente em paralelo) e quanto ao suporte de versões simplificadas de *Methods* (LINHARES, 2015).

Mesmo que veloz, paralela e com potencial descrição em alto nível, a abordagem em PON-HD possui certa limitação de escalabilidade, pois mapeia os elementos da cadeia de notificações de determinada aplicação para instâncias dos circuitos digitais correspondentes em *hardware*. Assim cada uma dessas instâncias consome uma determinada quantidade de elementos lógicos disponível na FPGA, limitando assim a quantidade de instâncias pela capacidade do dispositivo FPGA (KERSCHBAUMER *et al*., 2015; PORDEUS *et al*., 2016; KERSCHBAUMER, 2018; KERSCHBAUMER *et al*., 2018). Ainda, cada mudança na aplicação requer uma reconfiguração do FPGA, o que diminui a flexibilidade no uso desta abordagem em relação ao modelo em software*,* o qual têm seu código binário simplesmente carregado da memória para a plataforma em execução (LINHARES, 2015).

Neste contexto, por meio de uma solução híbrida, Peters (2012) propôs a implementação em lógica reconfigurável de um coprocessador PON (CoPON) inspirado no *Framework* PON VHDL Prototipal. A solução CoPON é uma forma na qual a parte da aplicação responsável pelo processamento facto-execucional é executada em um núcleo baseado em von Neumann e a parte da aplicação responsável pelo processamento (ou cálculo) lógico-causal e pela propagação de notificações é executada no coprocessador CoPON (PETERS *et al.*, 2012). A Figura 18 apresenta a cadeia de notificações com destaque para os componentes do PON (*Attributes, Premises, Conditions* e *Rules*) implementados em FPGA, no chamado co-processador PON.

Figura 18: Cadeia de Notificações do CoPON



Fonte: Peters *et al.*, 2012

O *hardware* do coprocessador é configurável de acordo com o projeto. Assim, a quantidade de instâncias de elementos PON (*i.e., Attributes*, *Premises*, *Conditions* e *Rules*) utilizados para a construção da aplicação PON pode ser configurada. Naturalmente, tal quantidade é limitada pelo tamanho de unidades lógicas disponíveis na FPGA. Os elementos *Methods* do PON, por sua vez, são codificados em software utilizando o *Framework* PON C++ 1.0 e são executados por um núcleo von Neumann sintetizado em FPGA (PETERS, 2012; LINHARES, 2015). Em tempo, essa solução poderia ser evoluída de maneira tal a ser sinérgica a desenvolvimento do PON em alto-nível com PON-YAML.

O modelo CoPON apresenta as limitações similares do *Framework* PON VHDL Prototipal e mesmo as evoluções desse. Isto com relação a escalabilidade e flexibilidade, pois o mecanismo de inferência é mapeado a nível de *hardware* em FPGA. Outra limitação é quanto à execução dos *Methods*, pois seu processamento é realizado em ambiente sequencial. Dessa forma, ainda que seja possível executar em um ambiente com múltiplos núcleos, cada um destes núcleos apresenta a complexidade de um núcleo *von Neumann* completo. Nesse contexto, o *Framework* PON VHDL Prototipal difere-se da solução por mapear esses elementos para circuitos relativamente simples para fins de prototipação (LINHARES, 2015).

Com o intuito de diminuir as deficiências apresentadas nos demais modelos PON já abordados, Linhares (2015) propôs uma nova abordagem para execução de software PON, na forma de uma arquitetura de computador denominada *Notification-Oriented Computer Architecture* (*NOCA* ou ARQPON - Arquitetura de Processador para o Paradigma Orientado a Notificações).

Esta arquitetura tem como objetivo implementar a dinâmica de notificações do PON de forma mais próxima de seu modelo teórico, permitindo que seja possível explorar suas características como a facilidade de paralelização. Além disso, o ARQPON objetiva flexibilidade, de tal forma que se torna possível alternar entre diferentes aplicações PON, somente pela substituição do software em memória, de forma similar ao que ocorre nos processadores atuais (LINHARES *et al.*, 2015).

A seu turno, Pordeus (2017) desenvolveu um simulador para o ARQPON, denominado de NOCASim, com o objetivo de executar as aplicações seguindo a *ISA* (*Instruction Set Architecture*) do ARQPON de forma semelhante ao protótipo implementado por meio de FPGA por Linhares *et al*. (2015). Tal simulador possibilita realizar avaliações com alto grau de paralelização, uma vez que não possui as limitações impostas pela quantidade de elementos lógicos disponíveis na FPGA. Isso permite que novas configurações da arquitetura possam ser exploradas e avaliadas em software com menor esforço do que quando implementadas diretamente em *hardware* (PORDEUS, 2017).

Contudo, embora os esforços de materializações do PON em *hardware* como o CoPON e o ARQPON tenham mitigado algumas deficiências do *Framework* PON VHDL Prototipal, uma solução efetiva que explorasse o potencial do PON integralmente sintetizado em *hardware* se mostrava promissora. Nesse âmbito, Kerschbaumer (2018) construiu uma versão melhorada do PON VHDL, denominada

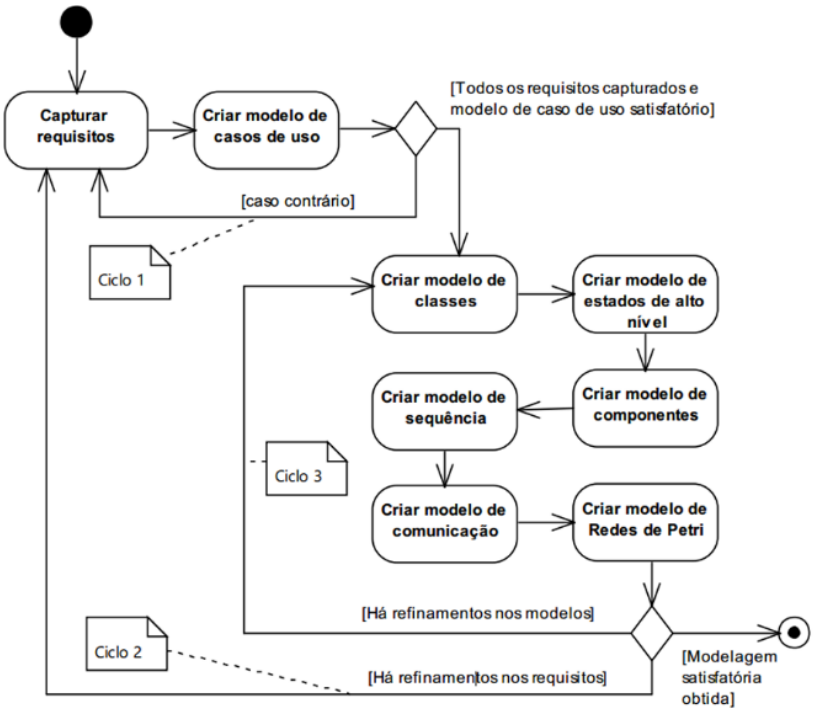
de PON HD 1.0. Esse novo framework é baseado em componentes minimalistas que simulam as entidades fundamentais do PON, apresentando maior nível de abstração em relação à abordagem de desenvolvimento puramente em VHDL (KERSCHBAUMER *et al*., 2015; PORDEUS *et al*., 2016, KERSCHBAUMER *et al*., 2018).

3.4.3 Engenharia de software para o PON

Segundo Wiecheteck (2011), embora o emprego convencional da UML e processos conhecidos no desenvolvimento de aplicações PON seja possível, as especificidades do PON motivam o uso mais particularizado de linguagens de modelagem e processos de software. Neste contexto, o DON (Desenvolvimento Orientado a Notificações) objetiva orientar projetistas na construção de projetos de software voltado ao PON (SIMÃO *et al*., 2012d).

Os modelos desenvolvidos no método são: modelo de classes, modelo de estados de alto nível, modelo de componentes, modelo de sequência, modelo de comunicação e modelo de Redes de Petri (MENDONÇA, 2015). O diagrama de atividades da Figura 19 ilustra os oito passos do método DON.

Figura 19: Atividades e ciclos do DON



Fonte: Mendonça *et al*., 2015

Em contrapartida, segundo Mendonça *et al*. (2015), o método DON abrange atividades relacionadas aos requisitos e análise e projeto de softwares em PON, sendo aderente aos processos tradicionais de desenvolvimento de software. Neste contexto, como ainda depende da UML, a análise realizada com DON não beneficia como um todo a descoberta das *Rules* no PON. Assim, o processo de identificação de *Rules* torna-se um intenso processo de síntese e requer muito esforço do desenvolvedor (MENDONÇA *et al*., 2015).

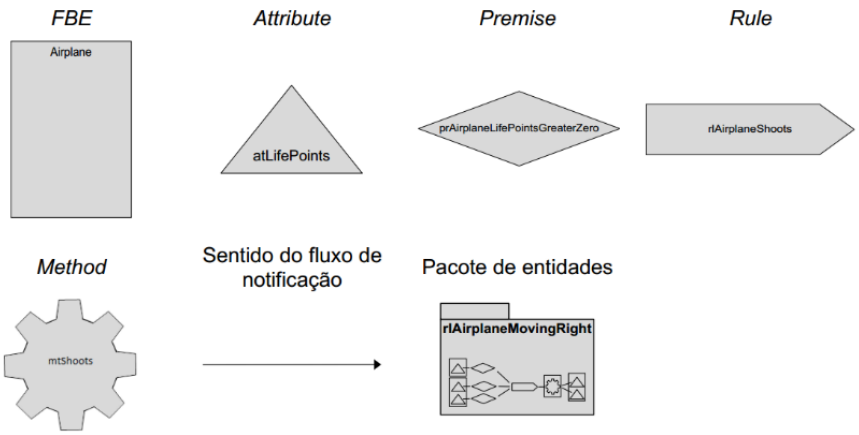
Visando amenizar tais deficiências do DON, Mendonça (2016) propôs uma Metodologia de Projeto de Software Orientado a Notificações (MON). Nessa metodologia, a orientação a regras, fatos e notificações é considerada desde os primeiros níveis da modelagem (MENDONÇA, 2016).

Ainda, em outro avanço relacionado ao desenvolvimento de software em PON, Kossoski (2015) propôs um método de teste para projetos de software que empregam o PON no seu desenvolvimento. Este método foi desenvolvido para ser aplicado nas fases de teste unitário e teste de integração.

Ainda, como parte deste método, Kossoski (2015) desenvolveu o Diagrama de Instâncias do PON. Neste contexto, o diagrama pode ser utilizado para representar as entidades que compõem o sistema. Assim, o desenvolvedor pode ter uma visão gráfica sobre o ciclo de notificações dos elementos envolvidos na aplicação.

Outrossim, segundo Kossoski (2015), caso seja necessário, o desenvolvedor poderá isolar elementos que realizam determinados processos no sistema, com o objetivo de facilitar análises específicas de fluxo de execução. A Figura 20 apresenta a notação dos elementos da cadeia de notificações PON utilizada no diagrama de instâncias PON.

Figura 20: Elementos do diagrama de instâncias do PON



Fonte: Kossoski, 2015

A correlação entre os elementos da cadeia de notificações e sua representação no diagrama de instâncias é dada por:

- *FBE* é representada como um objeto retangular.
- *Attribute* é representado como um triângulo.
- *Premise* é representada como um losango.
- *Rule* é representada com um objeto que desencadeia um fluxo de execução em um sentido. Considerou-se *Condition* como parte integrante da respectiva *Rule*.
- *Method* é representado como uma engrenagem.
- O sentido das notificações é representado como uma seta.

No âmbito da Engenharia de Requisitos, tanto para software quanto para sistemas em geral, alguns trabalhos tem sido realizados no sentido de discutir e propor uma abordagem para engenharia de requisitos de sistemas com base em uma abordagem orientada a regras e notificações para garantir a coerência e a compreensão efetivas desses requisitos ao longo do ciclo de vida de qualquer sistema complexo (SIMÃO *et al*., 2016; NOVAES *et al*., 2018).

3.4.4 Inteligência computacional e Inteligência Artificial

Desde os fundamentos que suportam o PON, estabelecidos na dissertação e na tese de doutorado de Simão (2001; 2005) e depois em Simão e Stadzisz (2009a), bem como em trabalhos de iniciação científica, mestrado, doutorado (e, até mesmo, disciplinas ligadas ao CPGEI e PPGCA da UTFPR), objetivaram desenvolver técnicas, tanto na área de software quanto na de hardware, que viabilizassem o desenvolvimento de aplicações segundo tal paradigma. Alguns destes foram nas chamadas áreas de Inteligência Artificial (IA) e Inteligência Computacional (IC) e até mesmo uma interseção destas (SIMÃO *et al*., 2003; SCHÜTZ *et al.,* 2018).

Banaszewski *et al*. (2007) apresentaram o PON como uma técnica de desenvolvimento de software baseada em conceitos de IA, além de desenvolver o chamado Framework PON C++ 1.0. Tal trabalho apoiou-se na premissa de que as entidades PON se comportam como *smart-objects* (objetos inteligentes). Nesse contexto, a mudança de um fato ativa apenas as avaliações estritamente necessárias por meio de colaboração via notificações de objetos ditos inteligentes.

Outro ponto do trabalho equipara o PON a sistemas baseados em agentes. Neste âmbito, o PON seria, em uma explicação assaz simplória e imprecisa, um tipo de evolução de Sistemas Baseados em Regras (SBR) e Orientação a Objetos (OO) que usa objetos inteligentes *Rules* como agentes e seus componentes colaborativos também como agentes. Por fim, aponta-se que as *Rules* detêm todo conhecimento lógico-causal relativo à aplicação, enquanto os *FBEs* detêm o conhecimento facto-execucional (BANASZEWSKI et. al., 2007).

Outro trabalho na área de IA/IC foi publicado por Melo *et al*. (2015) no âmbito de sua dissertação de mestrado, essa defendida subsequentemente e disponível em Melo (2016). Tal esforço visou a adaptação do PON para desenvolvimento de sistemas *Fuzzy*, inspirado em trabalhos embrionais prévios do grupo (SIMÃO *et al*., 2003; SOUZA *et al.,* 2009). Ademais, em Melo (2016), modificações no Framework PON C++ 2.0 foram realizadas, visando sua utilização no desenvolvimento de sistemas *Fuzzy*. Um exemplo de modificação foi a mudança na representação do estado lógico das *Premises*. Assim, o estado lógico passa a utilizar degraus de ativação, podendo assumir valores reais entre [0.0, 1.0] na versão *Fuzzy* do PON. Além disso, tais modificações permitem o desenvolvimento de sistemas híbridos, nos quais *Premises* do tipo *crisp* e do tipo *Fuzzy* podem compor a mesma *Rule* pelo uso de certos conectivos lógicos (MELO, 2016).

Em outro trabalho, este realizado por Schütz (2019) apresenta a aplicabilidade do PON na área de Redes Neurais Artificiais (RNAs). Em suma, o trabalho considera que notificações mimetizariam sinapses (entendendo que notificações se assemelham a sinapses em alguma ordem). Nesse processo, determinados conjuntos de *FBEs* associadas a certas *Rules*, constituiriam neurônios ou neuro-componentes factual-execucionais e outros ainda lógico-causais, os quais colaborariam por sinapses notificantes, formando o proposto modelo computacional NeuroPON (SCHÜTZ *et al*., 2015; SCHÜTZ *et al.,* 2018; SCHÜTZ, 2019).

3.5 PROPRIEDADES ELEMENTARES DO PON

O modelo computacional do PON apresenta uma estrutura particular, na qual as entidades reativas apresentam um acoplamento mínimo entre elas. Essa característica particular do paradigma previne ou, até mesmo, elimina a existência de redundâncias estruturais (*i.e.,* repetição de código) e temporais (*i.e.,* reavaliação

desnecessária de código) na execução de programas. Isto, supostamente, impacta positivamente no tempo de execução de programas e, principalmente, facilita a modularização e, portanto, a paralelização e distribuição do processamento (SIMÃO e STADZISZ, 2009; RONSZCKA *et al*., 2015).

Nesse âmbito, por meio desse modelo computacional explícito, o PON apresenta algumas propriedades elementares que são: (a) facilidades com o desenvolvimento pela orientação a regras em alto nível; (b) baixo tempo de processamento pela ausência de redundâncias e; (c) modelo preparado para uma execução paralela, e mesmo, distribuída em função do desacoplamento implícito dos elementos. De modo a apresentar as particularidades das propriedades elementares do PON, as subseções seguintes exploram as nuances de cada qual.

3.5.1 Facilidade de programação em alto nível orientado a regras

Conforme dito anteriormente, o PON encontra inspirações no PI/POO, tais como a flexibilidade algorítmica e a abstração em forma de classes/objetos de elementos facto-execucionais. Em relação ao PD, o PON aproveita conceitos da programação declarativa, como facilidade de programação em alto nível e a representação do conhecimento lógico-causal em regras. Desta forma, o PON provê a possibilidade de uso (de parte) de ambos os estilos de programação em seu modelo, ainda que os evolua no tocante ao processo de inferência ou cálculo lógico-causal (SIMÃO e STADZISZ, 2009; RONSZCKA *et al*., 2015).

Em suma, as primeiras materializações do paradigma, em forma de *frameworks*, possibilitaram a construção de elementos da base de fatos, em um formato orientado a classes/objetos derivados de uma classe genérica denominada *FBE*. Ademais, a base de regras tem sua composição constituída por instâncias da classe genérica *Rule*, bem como as demais classes que compõem o modelo computacional do PON. Nesse âmbito, na prática, o desenvolvedor constrói seus programas com base na criação dos elementos do modelo do PON, compondo *FBEs* e *Rules* em alto nível.

Ademais, em Ronszcka (2012) foi apresentado um conjunto de boas práticas para a programação em PON, tanto em termos de legibilidade e expressividade, quanto em termos de padronização para com a escrita de código em C++ voltado ao modelo do *Framework* PON C++ 2.0. Nesse mesmo trabalho foi proposto um conjunto

de macros (ou, pseudônimos i.e., instruções *define* do C/C++) para simplificar a composição de objetos do modelo, reduzindo o uso de instruções longas e repetitivas para tal.

A título de exemplo, o Código 6 apresenta as definições dos pseudônimos para os tipos de *Attributes* PON. No código ilustrado são apresentados apenas os tipos *Boolean* e *Integer*, porém tal recurso é aplicado igualmente a todos os demais tipos de *Attributes*.

Código 6: Definições de pseudônimos para os tipos de *Attributes* PON

```
1  #define BOOLEAN(fbe, attribute, value)
2     attribute =
3        SingletonFactory::getInstance()->createBoolean(fbe, value);
4     attribute->setName(#attribute)
5
6  . . .
7
8  #define INTEGER(fbe, attribute, value)
9     attribute =
10       SingletonFactory::getInstance()->createInteger(fbe, value);
11    attribute->setName(#attribute)
```

Fonte: Ronszcka, 2012

Outrossim, as demais entidades PON (*i.e., Premise*, *Condition*, *Rule*, *Action* e *Instigation*) também apresentam pseudônimos, porém suas composições são similares às demais definições apresentadas. O Código 7 apresenta um exemplo de criação de entidades PON utilizando as simplificações proporcionadas pelos pseudônimos.

Código 7: Exemplo de criação de entidades PON utilizando os pseudônimos

```
1  CHANGE_STRUCTURE(SingletonFactory::NOPVECTOR);
2
3  PREMISE(prAlarmOff, alarm->atOff, false, Premise::EQUAL);
4  PREMISE(prSirenOn, siren->atState, true, Premise::EQUAL);
5
6  METHOD(siren->mtTurnOff, siren->atState, false, Attribute::STANDARD);
7
8  RULE(rlTurnSirenOff, scheduler, Condition::CONJUNCTION);
9  rlFireAlarm->addPremise(prAlarmOff);
10 rlFireAlarm->addPremise(prSirenOn);
11 rlFireAlarm->addInstigation(siren->mtTurnOff);
```

Fonte: Autoria Própria

É possível observar que a aplicação dos pseudônimos na criação de entidades PON simplifica suas composições, evita redundâncias nas chamadas de

código e evita problemas com a integridade das mesmas (*i.e.,* nomes e ponteiros com conjunto de caracteres divergentes).

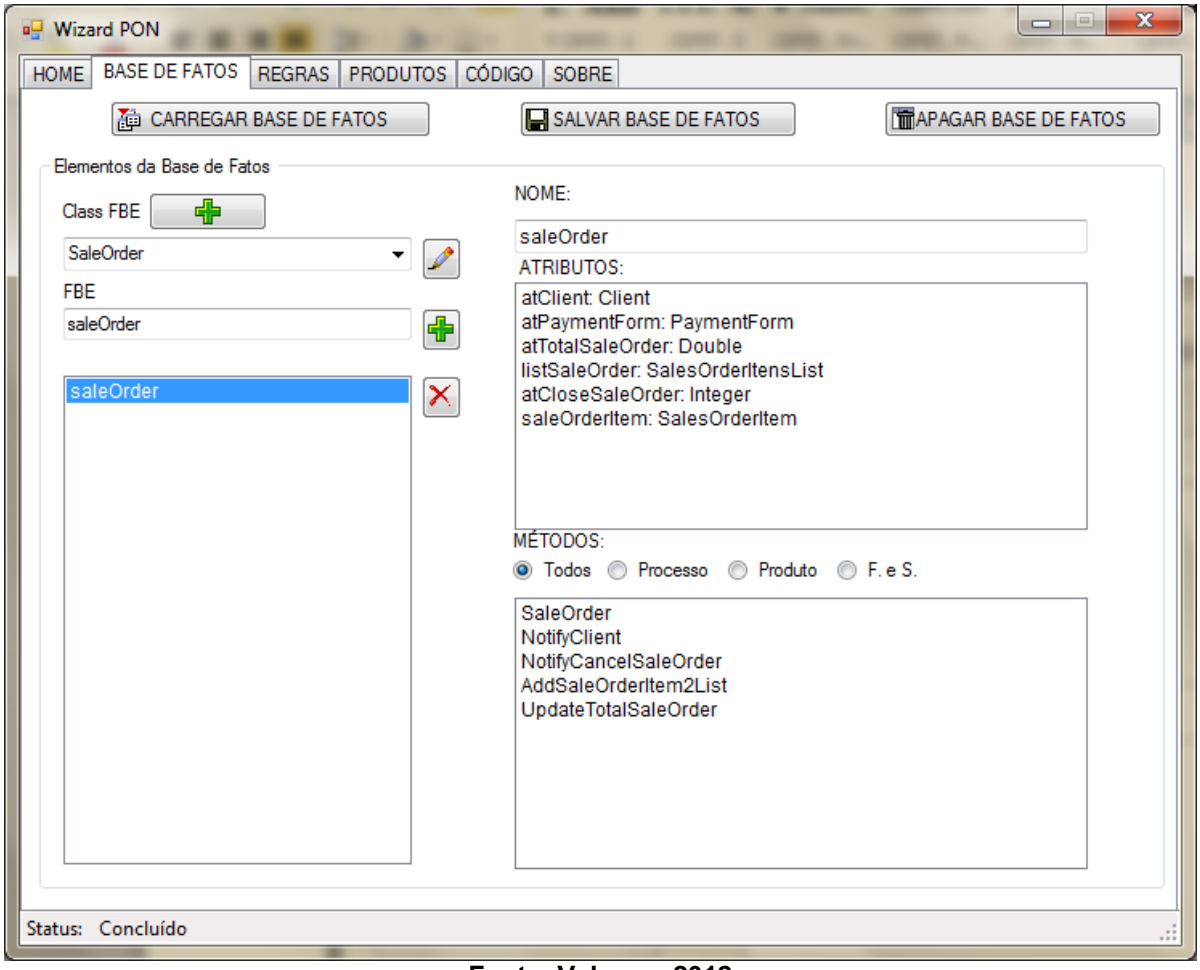
Outrossim, é importante ressaltar que o uso de pseudônimos apresenta um custo adicional na ‘montagem’ de uma aplicação PON, uma vez que realiza algumas chamadas de métodos adicionais. Entretanto, esse custo de processamento é relativamente baixo, sendo praticamente imperceptível na composição da maioria dos domínios de aplicações. Na verdade, na execução de uma aplicação propriamente dita, esse recurso não representa nenhum impacto no desempenho, uma vez que durante a execução de uma aplicação, caso nenhuma entidade adicional seja criada, tal recurso não é utilizado.

Além dos avanços relacionados a escrita de código para o Framework PON C++ 2.0, outro trabalho, este realizado por Valença (2012), apresentou uma ferramenta para a composição de programas de forma visual. Em suma, de modo a auxiliar no processo de desenvolvimento de programas PON, foi desenvolvida uma ferramenta específica para a geração de código para o *Framework* PON C++ 2.0. A principal finalidade desta ferramenta é facilitar o desenvolvimento e a composição de *FBEs* e *Rules* que sigam as diretrizes desta versão do *framework*.

A construção de um programa por meio dessa ferramenta *Wizard* acontece em alto nível, em uma ferramenta visual, com assistentes amigáveis. Esta ferramenta permite escrever a estrutura do programa que posteriormente tem suas classes ou instâncias geradas automaticamente, em um processo automatizado de geração de código.

A ferramenta é dividida em abas, separando o processo de criação de *FBEs* e *Rules*. A Figura 21 apresenta uma visão geral da ferramenta *Wizard* PON.

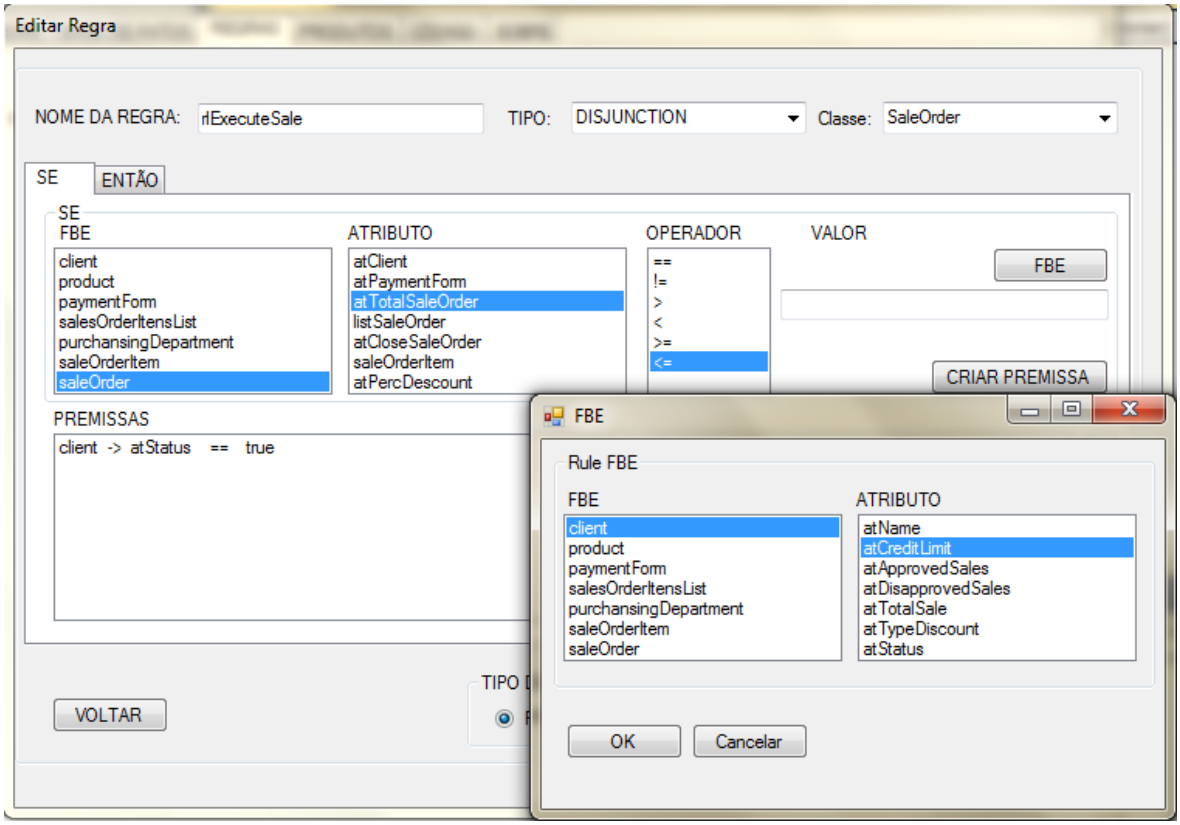
Figura 21: *Wizard* PON – Base de fatos (*FBEs*)



Fonte: Valença, 2012

Conforme apresentado na Figura 21, a ferramenta *Wizard* PON possibilita a criação e edição de *FBEs*, assim como também permite a importação de *FBEs* de outros programas. Outra característica importante da ferramenta é o vínculo dos *FBEs* com a criação das *Rules*. Esta etapa pode ser vislumbrada a partir da Figura 22.

Figura 22: *Wizard* PON – Criação de *Rules*



Fonte: Valença, 2012

Conforme apresentado na Figura 22, o processo de criação de programas PON é bastante intuitivo. O *Wizard* PON segue o padrão de construção orientado a base de fatos e regras, dos tradicionais motores de inferência. Isso facilita a adesão do paradigma pela comunidade em questão.

Em suma, a geração de código coleta todos os elementos criados e suas respectivas conexões e gera o código para *Framework* PON C++ 2.0. O conhecimento sobre como cada entidade específica deve se comportar é transparente para o programador, uma vez que é necessário apenas construir programas baseados em regras na ferramenta em questão.

Ademais, avanços outros relacionados com a facilidade de programação em alto nível, principalmente por meio de uma linguagem própria orientada a regras, é discutido em maiores detalhes no Capítulo 4 e 5 deste presente trabalho.

3.5.2 Baixo tempo de processamento - Cálculo Assintótico do PON

Em Simão (2005), há a análise assintótica do atualmente chamado Controle Orientado a Notificações (CON). Baseado nisso, Banaszewski (2009) em seu trabalho

apresenta uma análise assintótica do ION-PON. Ademais, tal análise assintótica do mecanismo de ION do PON é comparado para com os mecanismos de inferência pertinentes no domínio dos SBR, (precisamente o Rete, *TREAT*, *LEAPS* e *HAL*).

Tal comparação teve como objetivo avaliar teoricamente a complexidade de cada algoritmo à medida que o volume de dados de entrada aumenta. Segundo esta análise, o mecanismo de notificações do PON, o ION, é mais eficiente do que os algoritmos baseados em busca por co-relacionamento de dados e apresenta desempenho semelhante ao algoritmo do HAL ainda que ligeiramente melhor (BANASZEWSKI, 2009).

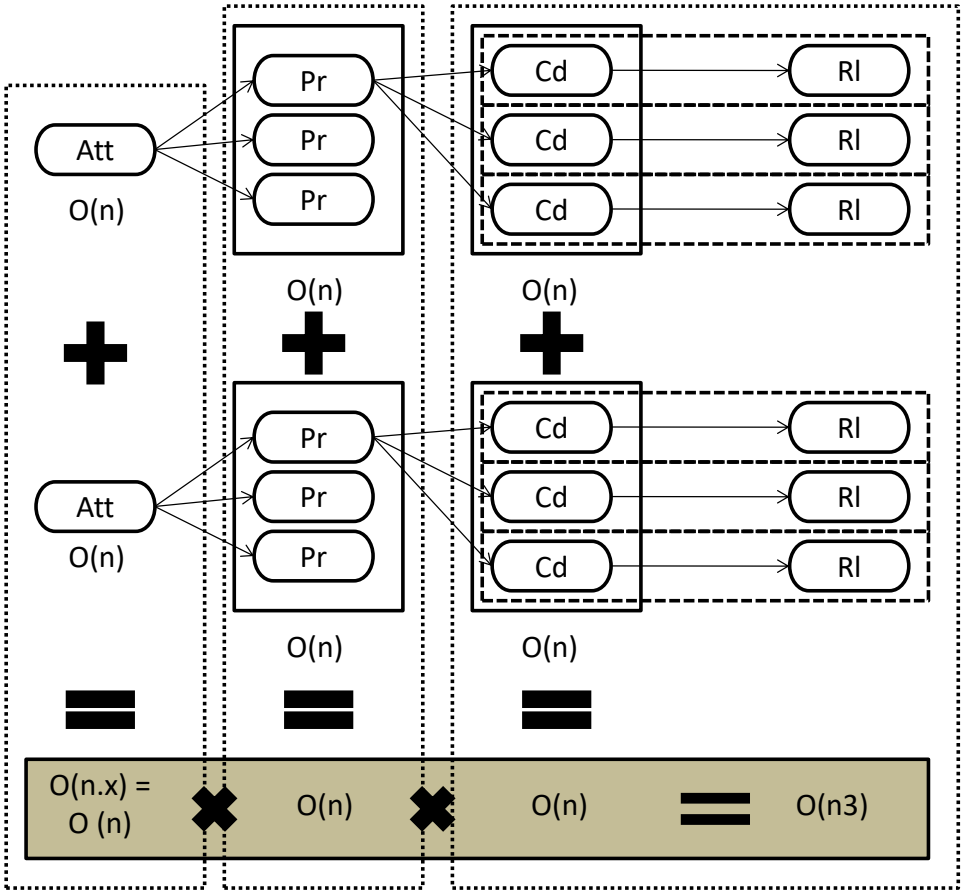
No ION-PON, a complexidade assintótica é polinomial no pior cenário, ou seja, é representada por $O(n^3)$ ou O(*FactBaseSize* * *nPremises* * *nRules*), onde *FactBaseSize* corresponde ao número máximo de objetos *Attributes*, *nPremises* corresponde ao número máximo de entidades *Premises* notificadas por estes *Attributes* e *nRules* corresponde ao número máximo de entidades *Conditions-Rules* notificadas por estas *Premises* (SIMÃO, 2005; BANAZEWSKI, 2009).

No pior cenário, a complexidade assintótica do ION-PON apresenta uma solução bastante similar ao mecanismo de notificações do algoritmo HAL. Ainda, a função temporal polinomial do HAL[7] ($O(n^5)$) se apresenta mais eficiente do que os algoritmos de inferência Rete, TREAT e LEAPS (BANAZEWSKI, 2009; RONSZCKA *et al.,* 2015).

A Figura 23 apresenta a análise assintótica do ION-PON, a qual demonstra as relações por notificações entre os objetos colaboradores, que também corresponde à quantidade de avaliações lógicas. Os *Attributes*, *Premises*, *Conditions* e *Rules* correspondem respectivamente aos símbolos com abreviações *Att*, *Pr*, *Cd* e *Rl*.

[7] Em relação ao *HAL*, apesar das grandes semelhanças, o mecanismo de notificações se diferencia na comunicação mais pontual entre os objetos. Enquanto que no *HAL* somente os componentes genéricos se comunicam, no mecanismo de notificações as próprias instâncias podem ser comunicar e de forma direta, evitando que entidades genéricas despendam buscas para relacionar as instâncias comunicantes (BANAZEWSKI, 2009; RONSZCKA *et al.*, 2015).

Figura 23: Cálculo assintótico do mecanismo de notificações



Fonte: Banaszewski, 2009

Entretanto, ao invés de se usar o pior caso, outra forma adequada para análise da complexidade polinomial do ION-PON é considerar o caso médio. Neste caso, a análise da complexidade é iniciada no começo do processo de notificação do ION-PON, através da entidade *Attribute*. Desta forma, as principais variáveis envolvidas em uma notificação de um *Attribute* são demonstradas na Figura 24.

A variável *NumPremises* é a soma do número de entidades *Premises* em relação ao seu respectivo *Attribute* e a variável *NumRules* é a soma do número de entidades *Rules* relacionadas a cada entidade *Premise* contada em *NumPremises*. Portanto, se for considerado simplesmente cada ciclo de inferência como a instigação de um Attribute e w como sendo o número de todos os *Attributes* existentes, uma média possível seria: *TMedium*(x) = (FBAT.1()+ … + FBAT.w()) / w. Assim, o resultado desta média seria uma constante, o que implicaria em uma complexidade linear O(n) para o ION-PON (SIMÃO, 2005; RONSZCKA, 2012).

Figura 24: Complexidade da Notificação da entidade *Attribute*

$$FB_{at}() = NumPremises + NumRules$$

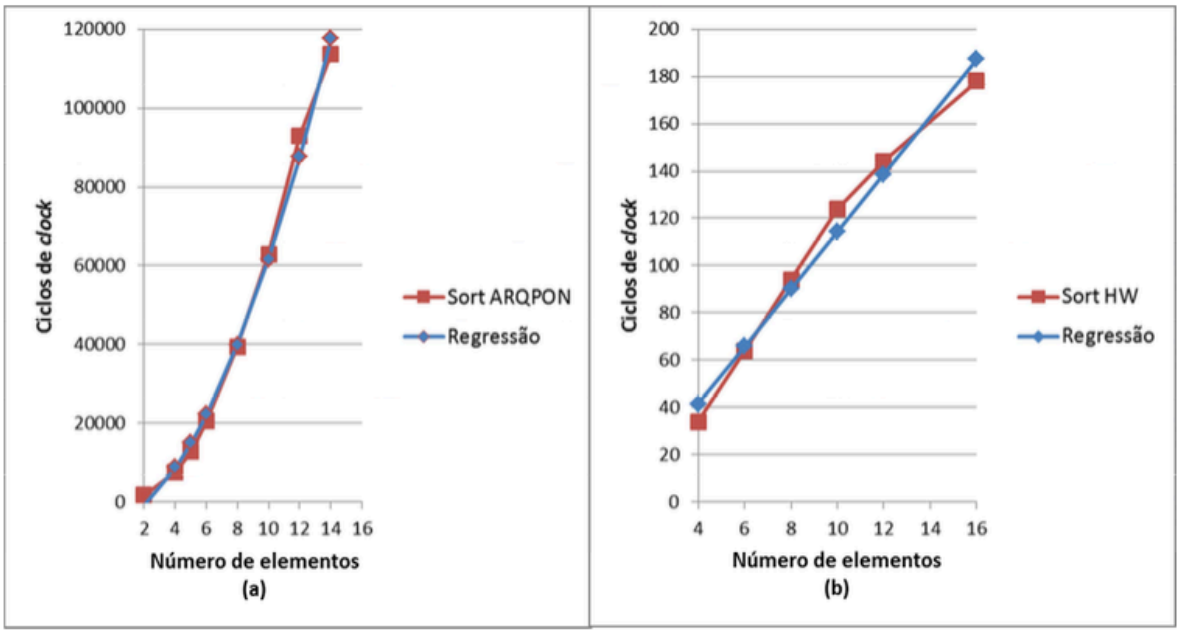
Fonte: Simão, 2005

Em Linhares (2015) foi apresentado um experimento para avaliar de forma preliminar o desempenho de soluções em hardware digital para o PON por meio de uma aplicação que realiza a ordenação de elementos numéricos dispostos em um *array*, com base em um algoritmo próprio. Tal aplicação, ao ser concebida segundo o PON, permite explorar o paralelismo de execução de elementos do modelo, uma vez que múltiplos pares de elementos adjacentes, do *array* de elementos a serem ordenados, podem ser processados simultaneamente (LINHARES, 2015).

Para fins de análise comparativa, a complexidade assintótica obtida a partir dos dados de desempenho foi comparada à complexidade assintótica obtida a partir de uma implementação específica do mesmo algoritmo sintetizado em FPGA. Esta implementação, diferentemente do protótipo da ARQPON, define e sintetiza elementos específicos de hardware para cada elemento do modelo de notificações, bem como canais de comunicação (notificação) exclusivos e específicos para execução da dinâmica da aplicação de ordenação.

Os dados de experimento preliminarmente obtidos e submetidos a regressão polinomial mostram que o desempenho do algoritmo de ordenação na ARQPON é de complexidade $O(n^2)$, onde n é o número de elementos a serem ordenados, conforme pode ser verificado na Figura 25(a). No entanto, a teoria do cálculo assintótico do PON e também os dados obtidos a partir de um experimento comparativo da implementação específica do algoritmo sintetizada em FPGA (Figura 25(b)) demonstram que tal algoritmo pode executar com complexidade O(n), dada a paralelização intrínseca do seu funcionamento dinâmico.

Figura 25: Experimentos com ARQPON e FPGA



Fonte: Linhares, 2015

Conforme apresenta a Figura 25, a plataforma ARQPON apresenta maior complexidade assintótica da implementação porque, na prática, a execução do algoritmo de ordenação nesta plataforma é sequencializada devido às operações de alocação e desalocação efetuadas pelo escalonador de *Rules*. Isto decorre do fato de que o protótipo não conta com um número suficiente de unidades de processamento (apenas 4 de cada tipo) para que todas as *Premises*, *Conditions* e *Methods* referentes aos possíveis pares de elementos adjacentes que poderiam ser ordenados simultaneamente sejam, de fato, executados em paralelo.

3.5.3 Modelo preparado para uma execução paralela/distribuída

Conforme apresentado nas subseções 3.4.2 e 3.5.2, as materializações do PON em hardware demonstraram a paralelização intrínseca do modelo computacional do PON. Os trabalhos já citados de Witt *et al*. (2011), Jasinski (2012) e Peters (2012). Todos estes trabalhos objetivaram viabilizar a implementação de parte (somente da cadeia de notificações) ou de toda uma aplicação PON diretamente em hardware, particularmente por meio da configuração de um dispositivo de lógica reconfigurável (*FPGA*).

Quanto à paralelização e distribuição via PON em software, alguns esforços são apresentados nesta subseção. Weber *et al.* (2010) especializou o *Framework*

PON C++ 1.0 (BANASZEWSKI, 2009) por meio da proposta de um ambiente *multithread* para execução concorrente dos elementos presentes no modelo de notificações do CON. Nesta as entidades CON (*Rule*, *Condition*, *Premise*, etc.) podem executar cada qual em uma *thread,* em particular. Contudo, tal solução não foi adicionada nas materializações atuais do PON de maneira explícita.

Belmonte *et al*. (2012) estenderam a implementação do *Framework* PON C++ 2.0, incluindo a execução de *Rules* por meio de *threads* independentes. Neste contexto, a extensão do *Framework* PON C++ 2.0 por meio de *threads*, intitulada de *Framework* PON C++ 3.0, permite ao desenvolvedor conceber software concorrente e também paralelo, este em ambiente *multicore* com SO apropriado, desde que informando qual parte do código precisa ser paralelizada (BELMONTE *et al*., 2016). Ainda, afim de efetuar o balanceamento dinâmico da carga de trabalho de software em ambientes *multicore*, foi criado o LobeNOI (*Load balancing engine for NOI – Notification-Oriented Inference – applications*) sobre o *Framework* PON C++ 3.0 (BELMONTE *et al*., 2016).

Assim, o chamado *Framework* PON C++ 3.0 (*threads* 3.0) em conjunto com a ferramenta baseada no método de paralelização LobeNOI fazem todos os cálculos e organizam o balanceamento da execução do software automaticamente, sem a intervenção explícita do desenvolvedor. Por fim, resultados obtidos demostraram que há melhora no aproveitamento da capacidade de processamento disponível na plataforma estudada que utiliza múltiplos núcleos (*multicore*) (BELMONTE *et al*., 2016).

Por fim, além da paralelização, outros trabalhos com relação à distribuição podem ser citados. Utilizando protocolos de rede (TCP/UDP – *User Datagram Protocol*) e por meio de W*eb services*, alguns destes esforços foram iniciados em disciplinas[8] e naturalmente evoluíram para pesquisas independentes, as quais em alguns casos, se consolidaram em publicações (OLIVEIRA, 2016; TALAU, 2016; BARRETO, 2016; BARRETO *et al*., 2018; OLIVEIRA *et al.,* 2018).

[8] Disciplina: Tópicos Avançados Em Sistemas Embarcados. Tema: Paradigma Orientado a Notificações. Código CASE102. Programa PPGCA/UTFPR - Prof. Jean Marcelo Simão e Prof. João A. Fabro. 1º Trimestre de 2016. & Disciplina: Tópicos Especiais Em EC: Paradigma Orientado a Notificações. Código TEC0301 (PGEID/PGEIM). Programa. CPGEI/UTFPR - Prof. Jean Marcelo Simão. 1º Trimestre de 2016.

Pode-se afirmar, portanto, que estes esforços tratados nessa subseção corroboram para demonstrar o PON como uma solução desacoplante, permitindo paralelização e mesmo distribuição. Esses resultados de paralização e mesmo distribuição, até certo ponto, são confirmados nas implementações do PON em hardware.

3.6 CONCEITOS E FUNCIONALIDADES DO PON

Além da estrutura geral do PON, organizada por meio do modelo de entidades reativas e notificantes, um conjunto de conceitos e funcionalidades são essenciais para a organização e execução adequada do fluxo de notificações. Nesse âmbito, as subseções subsequentes exploram os detalhes dos principais conceitos e funcionalidades do PON.

3.6.1 Mecanismo de escalonamento de *Rules* e Resolução de conflitos

De maneira geral, uma das principais características do PON é a execução de forma desacoplada (podendo ser inclusive paralela se a arquitetura permitir) dos elementos de seu modelo. Nesse âmbito, uma questão importante ao PON, é a identificação e resolução de possíveis conflitos que a execução desacoplada pode originar (PORDEUS, 2017).

Basicamente, um conflito ocorre quando pelo menos duas entidades diferentes em execução (*e.g.,* duas ou mais *Rules*) dependem de um mesmo elemento/recurso compartilhado (*e.g.,* um dado) em um mesmo instante de tempo, de forma que o elemento compartilhado em questão deve ser utilizado exclusivamente por apenas uma das entidades, naquele instante de tempo, originando um conflito (SIMÃO; STADZISZ, 2010).

Por exemplo, para o controle de um robô, constitui-se em um conflito a situação de duas *Rules* aprovadas quando uma *Rule* almeja que um dado *FBE Robot* vá para a direita e outra *Rule* almeja que ele vá para *esquerda*. Nesse caso, o *FBE Robot*, seu *Attribute Status* e seu *Method Move* são compartilhados e exclusivos. No caso, do conflito dado, alguma estratégia tem de ser fornecida para evitar o conflito (PORDEUS, 2017). Ademais, a estratégia de solução de conflito deveria (ou ao menos

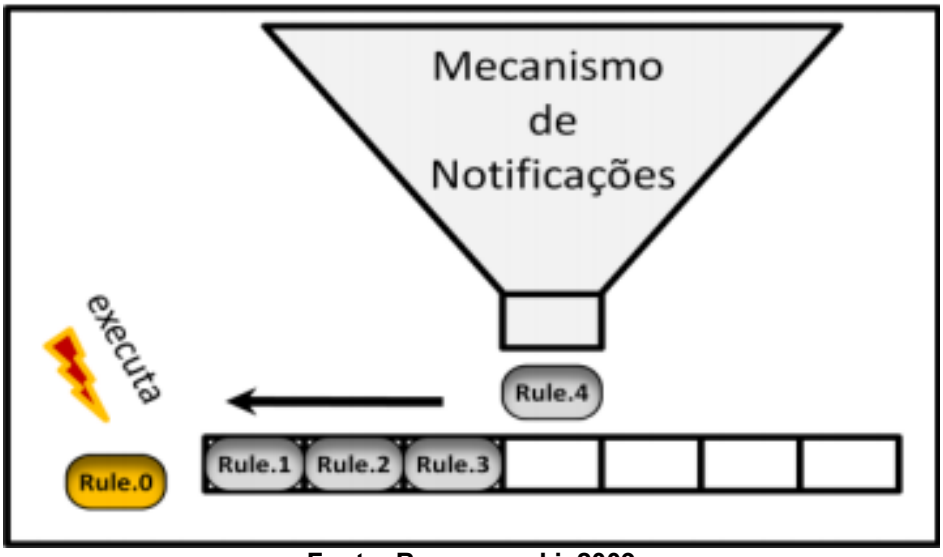
poderia) ser determinística no sentido de tornar sempre a mesma decisão em um mesmo contexto dado.

Além das questões de conflito determinístico, outra questão correlata e pertinente ao PON é a garantia de determinismo, ao menos o aqui chamado de determinismo de evolução. Nesse sentido, uma execução determinística significa que todas as entidades dedicadas às relações causais (como as *Rules*, *Conditions* e *Premises* no caso do PON) devem ter a mesma oportunidade de avaliar mudanças de estados em entidades pertinentes (SIMÃO e STADZISZ, 2010). No caso do exemplo acima, ambas as *Rules* precisariam ter uma oportunidade equivalente de avaliação e eventual aprovação em função da mudança de estado do *Attribute Status* do *FBE Robot*, garantindo assim um comportamento determinístico na evolução da avaliação lógico causal (PORDEUS, 2017).

Devido a este fato, o PON depende de um mecanismo de resolução de conflitos determinísticos e garantia de determinismo de evolução. Tal mecanismo de resolução de conflitos, em particular, pode ser implementado de diferentes formas. Em todo caso, entretanto, a sua essência é controlar o fluxo de notificações a partir dos elementos do modelo do PON (*Attributes*, *Premises* e *Conditions*) definidos por cada *Rule* aprovada, de maneira que as *Actions* e *Instigations* pertinentes relativas a *Methods* exclusivos (*i.e.,* que modificam *Attributes* exclusivos) não ocorram de forma conflitante com outras *Rules* também aprovadas (FERREIRA, 2015; LINHARES, 2015).

Para ambientes monoprocessados e monoprocesso, Banaszewski (2009) propôs um escalonador de *Rules* formado por uma estrutura de dados do tipo linear (*e.g.,* pilha, fila ou lista). Essas estruturas guardam referências para as *Rules* aprovadas, conforme ilustra a Figura 26. Assim, tais estruturas recebem as *Rules* na ordem em que elas são aprovadas, podendo reorganizá-las de acordo com os preceitos de cada estratégia adotada (BANASZEWSKI, 2009). Nesse contexto monoprocessado e monoprocesso, a solução naturalmente leva a resolução de conflito determinístico.

Figura 26: Modelo Centralizado de Resolução de Conflitos



Fonte: Banaszewski, 2009

Nesse sentido, conforme a estratégia de resolução de conflitos pré-determinada pelo desenvolvedor, as *Rules* em questão serão efetivamente executadas. Nesse âmbito, os modelos de resolução de conflitos empregados para o PON em ambientes monoprocessados são:

- ***BREADTH***: se baseia no escalonamento *First In, First Out (FIFO)*, ou seja, refere-se à execução de entidades *Rule*, seguindo uma estrutura de dados do tipo fila*;*
- ***DEPTH***: se baseia no escalonamento *Last in, First Out (LIFO)*, ou seja, refere-se à execução de entidades *Rule*, seguindo uma estrutura de dados do tipo pilha;
- ***PRIORITY***: organiza as entidades *Rule* de acordo com as prioridades definidas nas mesmas; e
- ***NO_ONE***:, as *Rules* são aprovadas e executadas imediatamente, não utilizando o mecanismo de escalonamento.

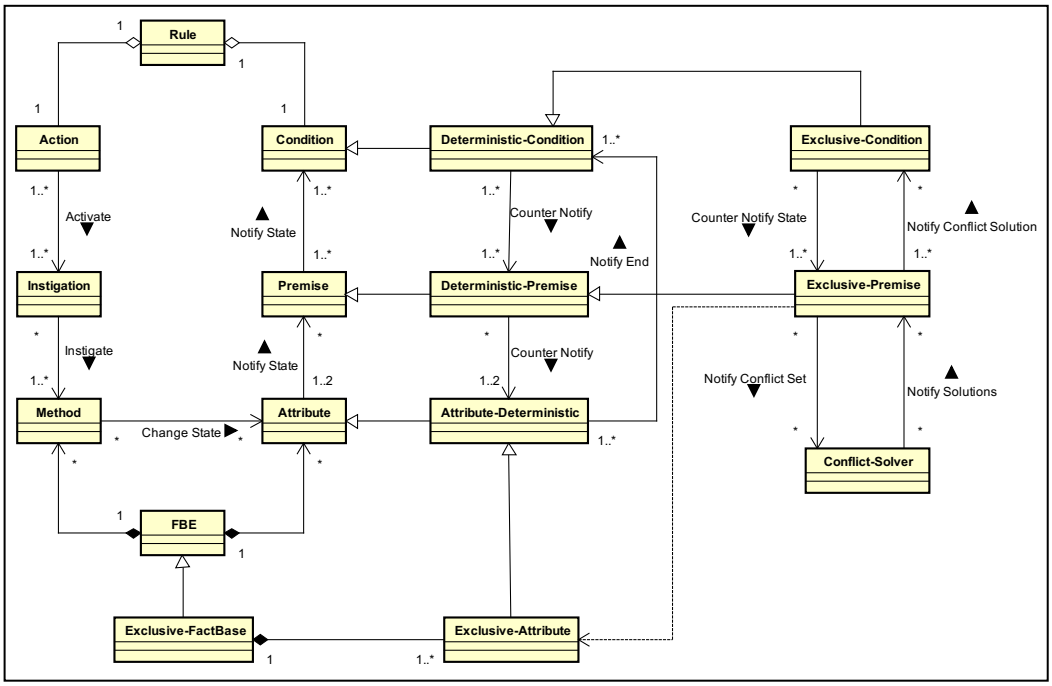
Basicamente, a definição do tipo da resolução de conflito adotada pelo desenvolvedor implica no método utilizado para a execução das *Rules* da aplicação. A definição de resolução de conflitos de *Rules* deve ser adotada, principalmente, em aplicações distribuídas ou concorrentes.

As soluções para evitar os conflitos apresentados são particularmente aplicáveis a soluções PON monoprocessadas, ainda que até possam ser úteis em soluções multiprocessadas e distribuídas (BANASZEWSKI, 2009). Entretanto, na verdade, a definição de resolução de conflitos de *Rules* deve ser adotada em aplicações concorrentes ou distribuídas com soluções que lhe sejam mais apropriadas. É importante salientar que as versões 1.0 e 2.0 do *Framework* PON C++ implementam estes modelos de resolução de conflitos proposto por Banaszewski (BANASZEWSKI, 2009; VALENÇA, 2012; RONSZCKA, 2012).

Com relação a ambientes multiprocessados, em (SIMÃO, 2005) é apresentada uma solução para o CON e em (SIMÃO e STADZISZ, 2010) é apresentada uma solução para o PON. Basicamente, em tais trabalhos, é apresentada uma estratégia de resolução de conflitos e garantia de determinismo baseada em: sincronização de acesso aos elementos PON. Para tal houve especializações de seus elementos em *Exclusive-Attribute*, *Exclusive-Premise*, *Exclusive-Condition*, *Exclusive-FactBase*, *Attribute-Deterministic*, *Premise-Deterministic* e *Condition-Deterministic*, os quais, em suma, se baseiam na confirmação de recebimento de notificações.

Isto se constitui basicamente de contra notificações (notificações no sentido contrário). De modo geral, tais contra notificações permitem indicar ao emissor que recebeu uma notificação, viabilizando que o emissor da notificação tome decisões em relação a resolver eventuais conflitos entre múltiplas *Rules* que foram aprovadas enviando notificação ao ganhador do conflito. Isto também serve para algum nível de garantia de determinismo conforme descrito em (SIMÃO; STADZISZ, 2010) e em (LINHARES, 2015). A Figura 27 apresenta modelo de notificações do PON para resolução de conflitos e determinismo proposto por Simão e Stadzisz (2010).

Figura 27: Metamodelo de contra-notificações do PON



Fonte: Adaptado de Simão e Stadzisz, 2010

Em Linhares (2015) foi apresentado um mecanismo escalonador/resolvedor de conflitos para o ARQPON. Em tal solução o conceito de contra notificações pode ser aplicado de forma implícita. No modelo conceitual, a confirmação de uma única *Exclusive-Condition* de um grupo de *Conditions* conflituosas é conceitualmente efetuada por meio de uma contra notificação para a *Exclusive-Premise* comum e a decisão de resolução é realizada por esta mesma *Premise*. Na ARQPON esta confirmação pode ser realizada pelo mecanismo sem a necessidade de uma contra notificação explícita, dado que ele tem acesso (via *snooping* da Rede de notificação *Premise-Condition*) a todo o fluxo de notificações de *Premises* bem como também tem acesso (via *snooping* da Rede de notificação *Condition-Method*) ao resultado lógico de todas as *Conditions* aprovadas. Sendo assim, o mecanismo pode confirmar a propagação do resultado lógico “verdadeiro” somente da *Condition* que for vencedora da resolução de conflito, sendo o critério de resolução também de responsabilidade do mecanismo (LINHARES, 2015).

3.6.2 Compartilhamento de entidades PON

Em suma, um programa em PON é desenvolvido por meio da definição de um conjunto de entidades do modelo do paradigma, organizadas de maneira a criar uma

cadeia de notificações coerente para a solução de um problema específico. Nesse caso, uma boa organização das entidades se faz necessário para criar um programa coeso e bem estruturado. Nesse âmbito, é considerado uma boa prática compartilhar entidades no PON, tanto em questões de facilidade de desenvolvimento, quanto em questões de desempenho. O correto uso do compartilhamento de entidades PON eliminaria a criação de entidades redundantes, bem como possíveis notificações desnecessárias para essas (RONSZCKA, 2012).

Ademais, o compartilhamento de entidades pode acontecer de forma manual, na qual o desenvolvedor deveria se preocupar com a correta estruturação do programa (*i.e.,* como no caso dos *Frameworks* PON) ou, ainda, poderia acontecer de forma automática. Nesse âmbito, o sistema "montador" organizaria as entidades e eliminaria elementos redundantes. De modo a exemplificar o uso de tal boa prática no *Framework* PON C++ 2.0, considera-se um cenário hipotético no âmbito do sistema de alarme. Em tal cenário, conjuntos de sensores são agrupados em setores, no qual cada setor poderia utilizar conhecimento compartilhado do sistema. De modo a ilustrar um exemplo desse cenário, o Código 8 apresenta um trecho de código que contempla o compartilhamento de entidades no *Framework* PON C++ 2.0.

Código 8: Exemplo de compartilhamento de entidades PON

```
1  PREMISE(prAlarmOn, alarm->atOn, true, Premise::EQUAL);
2
3  . . .
4
5  METHOD(mtFireSiren, siren->atOn, true, Attribute::STANDARD);
6
7  . . .
8
9  RULE(rlSectorAFired, scheduler, Condition::CONJUNCTION);
10 rlSectorAFired->addPremise(prAlarmOn);
11 rlSectorAFired->addPremise(prSensor1Fired);
12 rlSectorAFired->addPremise(prSensor2Fired);
13 rlSectorAFired->addInstigation(mtFireSiren);
14
15 RULE(rlSectorBFired, scheduler, Condition::CONJUNCTION);
16 rlSectorBFired->addPremise(prAlarmOn);
17 rlSectorBFired->addPremise(prSensor3Fired);
18 rlSectorBFired->addPremise(prSensor4Fired);
19 rlSectorBFired->addInstigation(mtFireSiren);
20
21 RULE(rlSectorCFired, scheduler, Condition::CONJUNCTION);
22 rlSectorCFired->addPremise(prAlarmOn);
23 rlSectorCFired->addPremise(prSensor5Fired);
24 rlSectorCFired->addPremise(prSensor6Fired);
25 rlSectorCFired->addInstigation(mtFireSiren);
```

Fonte: Autoria Própria

Conforme ilustrado no Código 8, a primeira *Premise* (linha 1) é compartilhada com três *Rules* distintas (nas linhas 10, 16 e 22). Nesse caso, quando tal *Premise* ter seu estado ‘aprovado’, ela irá notificar todas as *Rules* interessadas, otimizando assim o processo ao evitar entidades e notificações redundantes. Além disso, o compartilhamento também pode acontecer com a entidade *Method*, a qual pode ter sua execução instigada por mais de uma *Rule* do programa (conforme linhas 13, 19 e 25). Ao bem da verdade, no PON, à luz de sua teoria, todas as entidades do modelo poderiam ser compartilhadas, principalmente, se isto ocorrer de forma automática.

Outrossim, o artigo (SIMÃO *et al*., 2012c) apresenta comparativos quantitativos entre duas implementações de um simulador para o clássico jogo Pacman. Apesar de não estar explicito o impacto do compartilhamento de *Premises* nos resultados apresentados nesse artigo, devido a outras questões como melhorias no *framework* e mudanças no ambiente de testes, foi possível observar que o modo como a aplicação foi reimplementada teve grande impacto no desempenho. Valença (2012) demonstrou que a implementação do *Framework* PON C++ 2.0 apresentou ganhos de desempenho de cerca de 2 vezes em média em relação a versão 1.0. No artigo (SIMÃO *et al*., 2012c), em especial, os ganhos para com a versão anterior se mostraram ainda mais favoráveis, apresentando um desempenho cerca de 5 vezes superior. Nesse caso, mostra-se que o cuidado com o compartilhamento de entidades, entre outras boas práticas, é essencial para atingir eficiência no PON (RONSZCKA, 2012).

3.6.3 Regras de Formação – *Formation Rules*

Outro conceito proposto ao PON foi o de *Formation Rules* (Regras de Formação), algo que também surgiu no Controle Orientado a Notificações (CON) (SIMÃO, 2001; SIMÃO *et al*., 2003). Em suma, cada *Formation Rule* permite a criação de *Rules* específicas, a partir da representação genérica de uma *Rule* (PORDEUS, 2017; SANTOS, 2017). Este conceito é útil quando o conhecimento causal de uma *Rule* é comum para diferentes conjuntos de instâncias de *FBEs*, ou seja, um conjunto de *Rules* específicas se diferencia apenas nas combinações das instâncias referenciadas.

A título de exemplo, considera-se um cenário hipotético de um sistema de alarme, no qual todos os usuários administradores do sistema teriam acesso a todas

as centrais de alarme conectadas. Nesse cenário, considerando uma ação de avisar cada usuário administrador sobre eventuais disparos em cada uma das centrais de alarme conectadas, seria necessário replicar o conhecimento dessa condição para cada combinação de usuários e centrais de alarme. No caso de um cenário com um número usuários e centrais de alarme, sem a utilização do conceito de *Formation Rules* seria necessário replicar as *Rules* manualmente para cada instância declarada, tornando o processo de desenvolvimento trabalhoso e tendente a erros.

Com o uso das *Formation Rules*, por outro lado, o conhecimento lógico-causal dessa *Rule* especial é informada de forma genérica, com base no modelo da classe de *FBE User* e *FBE Central*, em vez de instâncias pontuais de cada qual. Assim, em tempo de montagem do programa, cada combinação de instâncias de *User* e de *Central* (*e.g., user1 x central1, user1 x central2, user2 x central1, user2 x central2*), teriam instâncias pontuais e específicas dessa *Rule* genérica, formando uma composição de *N* x *M* instâncias.

Com base nesse conceito, o desenvolvedor não precisa informar manualmente as características necessárias para a criação de cada *Rule* pontual. Como elas apresentam o mesmo conhecimento lógico-causal, tornar a replicação de *Rules* uma tarefa automática é desejável e minimiza erros, além de promover maior legibilidade do código em geral, uma vez que o código final da aplicação será composto apenas pela *Rule* genérica ao invés de *N* cópias de *Rules* similares.

3.6.4 Propriedades reativas dos *Attributes*

Em linhas gerais, os *Attributes* proporcionam grande impacto na concepção de aplicações PON, uma vez que representam o ponto de partida para a realização do cálculo lógico-causal do PON por meio de suas propriedades reativas. Nesse sentido, alguns cuidados com a criação deveriam ser tomados para atingir um nível de programação eficiente e correta.

De maneira geral, um *Attribute* pode apresentar três comportamentos distintos ao ter seu estado alterado. O comportamento padrão, de acordo com a teoria, é notificar as *Premises* interessadas somente quando o estado de tal *Attribute* tiver sofrido alterações. Todavia, existem situações que demandam que uma *Rule* seja reavaliada e executada novamente, mesmo quando os estados de suas entidades colaboradoras não tenham sofrido alterações, necessitando de uma renotificação para

este fim. Ainda, existem casos em que as alterações no estado de um *Attribute* não deveriam gerar notificações em nenhuma das situações (RONSZCKA, 2012).

As alterações no estado de um *Attribute* ocorrem normalmente na execução de um *Method*. Nesse âmbito, para definir o modo com que um *Attribute* vai reagir perante a alterações em seu estado, o Código 9 apresenta exemplos de código no *Framework* PON C++ 2.0, utilizando as três possíveis *flags* de modificação de comportamento.

Código 9: Exemplo de compartilhamento de entidades PON

```
1 METHOD(mtActivateAlarm, alarm->atOn, true, Attribute::STANDARD);
2
3 METHOD(mtReset, accessControl->atCount, 0, Attribute::RENOTIFY);
4
5 METHOD(mtFireSiren, siren->atOn, true, Attribute::NO_NOTIFY);
```

Fonte: Autoria Própria

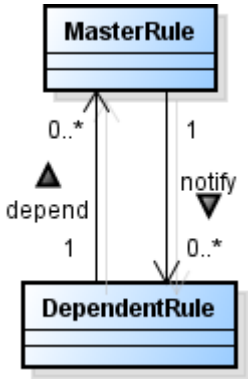
Conforme ilustrado no código do Código 9, a linha 1 representa a atribuição padrão de um valor para um dado *Attribute*, o que resultaria no disparo do fluxo de notificações caso o estado desse apresente um valor diferente do anterior. A linha 3, por sua vez, representa a atribuição baseada em renotificações, disparando o fluxo de notificações em todas as ocasiões, independente do estado anterior do *Attribute* em questão. A linha 5, particularmente, apresenta a atribuição para casos em que o disparo do fluxo de notificações seja indesejado. Outrossim, caso a *tag* seja omitida, o comportamento padrão será adotado.

3.6.5 Dependência entre *Rules* – *Master Rule*

O conceito denominado de dependência entre *Rules* permite a criação de *Rules* que não são ativadas apenas pelo mecanismo de notificação convencional (*i.e.,* por meio de uma *Condition* com valor lógico verdadeiro), mas pela ativação por meio do mecanismo de notificações convencional somada a aprovação de outra *Rule* da qual depende, a qual passa a ser chamada de *Master Rule*. Nesse sentido, as *Rules* dependentes atuam na redução de notificações geradas pelas *Premises* e *Conditions* que são compartilhadas por múltiplas *Rules*. Quando as entidades *Premises* e *Conditions* em questão forem aprovadas, estas direcionariam suas notificações apenas a uma única *Rule* (*Master Rule*) que, por sua vez, notifica as demais *Rules* dependentes (RONSZCKA, 2012).

No âmbito de eficiência na execução, tais *Rules* atuariam na redução de notificações geradas pelas *Premises* em questão, que direcionariam suas notificações apenas à *Master Rule*. Em relação à facilidade de composição de aplicações, tal abordagem simplificaria a essência de todas as demais *Rules*, tornando o código mais legível e consequentemente mais manutenível (RONSZCKA, 2012). A Figura 28 exemplifica a estrutura da dependência de *Rules*.

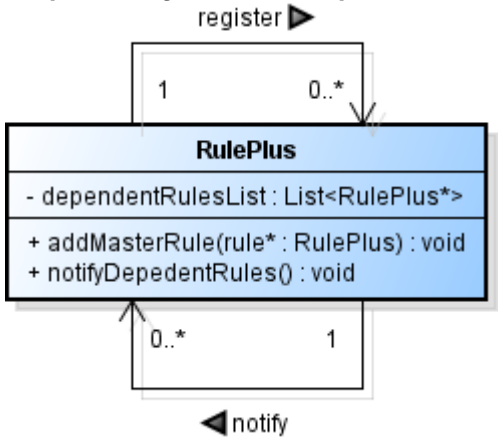
Figura 28: Dependência entre *Rules*



Fonte: Ronszcka, 2012

Conforme ilustra a Figura 28, a dependência entre *Rules* é formada por meio de uma associação entre a classe dependente e a classe mestre. A classe mestre, por sua vez, notifica mudanças em seu estado para a(s) classe(s) dependente(s). É importante ressaltar que esse diagrama ilustra apenas conceitualmente a maneira como essa dependência é formada (RONSZCKA, 2012). No modelo/codificação real do *Framework* PON C++ 2.0, ambas as funcionalidades de tais classes foram definidas e implementadas na classe *RulePlus*, conforme ilustra a Figura 29.

Figura 29: Representação real da dependência entre *Rules*



Fonte: Ronszcka, 2012

Conforme ilustrado na Figura 29, a classe *RulePlus* apresenta em seu âmbito ambas as funcionalidades. Neste sentido, uma *RulePlus* poderia atuar tanto como uma *Rule* dependente quanto como uma *Rule* mestre, inclusive ao mesmo tempo. Essa funcionalidade apresenta as características do padrão de projeto *Observer*, sendo esse então aplicado na estrutura dessa classe. Neste âmbito, a classe possibilitaria a adição de entidades interessadas em seu estado (*Rules* dependentes) que seriam notificadas a cada mudança de estado ocorrido na *Rule* mestre (*Rule* aprovada/desaprovada) (RONSZCKA, 2012). De modo a facilitar o entendimento da utilização dessa funcionalidade, o Código 10 exemplifica o uso dela na composição de *Rules* no *Framework* PON C++ 2.0.

Código 10: Exemplo de utilização da funcionalidade de dependência de *Rules*

```
1  RulePlus* rlAlarmSystemOn(Condition::CONJUNCTION);
2  rlAlarmSystemOn->addPremise(prAlarmOn);
3  rlAlarmSystemOn->addPremise(prSensorEnabled);
4
5  RulePlus* rlFireAlarm(Condition::CONJUNCTION);
6  rlAccelerateTurbines->addMasterRule(rlAlarmSystemOn);
7  rlAccelerateTurbines->addPremise(. . .);
8  rlAccelerateTurbines->addInstigation(. . .);
```

Fonte: Autoria Própria

Conforme apresenta o trecho de código do Código 10, a segunda *Rule* utiliza o método *addMasterRule* (linha 6) e passa como referência a primeira *Rule* (*i.e., rlAlarmSystemOn*), com a finalidade de definir uma dependência entre a primeira e a segunda *Rule* do exemplo em questão. Acrescentando, a *Rule rlFireAlarm* representa uma *Rule* dependente, enquanto que a *Rule rlAlarmSystemOn* representa respectivamente a *Rule* mestre dessa.

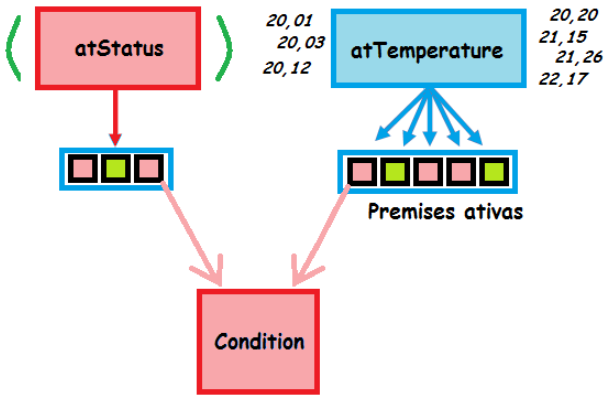
Outrossim, ao ser criado um relacionamento de dependência entre duas *Rules*, a *Rule* dependente receberia internamente em sua estrutura um vínculo com uma *Premise* adicional, por meio de uma *SubCondition*. A mudança do estado lógico de tal *Premise* (provocado pela *Rule* mestre ao ser executada) faz com que o ciclo de notificações seja instigado. Deste modo, caso as demais *Premises* de uma *Rule* dependente apresentem estado lógico verdadeiro, a execução dessa *Rule* é ativada, dando prosseguimento a sua execução (RONSZCKA, 2012).

3.6.6 Entidades impertinentes

De maneira geral, a reatividade presente nos *Attributes* proporcionaria uma execução livre de avaliações redundantes e desnecessárias, comuns aos paradigmas de programação usuais (cf. Seção 2.1). Entretanto, existem casos em que a variação de um *Attribute* encadeia sequencias de notificações indesejáveis. Isso normalmente ocorre em situações que um dado *Attribute*, apesar de não ser determinante para a aprovação de uma *Rule* em um dado contexto, apresenta constantes mudanças de estado, disparando o fluxo de notificações a cada variação em seu estado. Assim, tais notificações desnecessárias impactariam negativamente no desempenho de execução de uma aplicação PON (RONSZCKA, 2012).

A título de exemplo, considere o cenário hipotético de um sistema de refrigeração, no qual pequenas mudanças em um *Attribute* referente à temperatura interna (*atTemperature*) de uma casa que seria pertinente a uma dada *Rule*. Esta *Rule* seria responsável por ativar o ar condicionado se a temperatura interna atingir um dado valor e se alguém estiver na casa (*atStatus true*). Entretanto, a casa passa maior parte do tempo vazia (*atStatus false*). Assim, a maior parte das notificações do *Attribute* em questão (*atTemperature*) seriam 'impertinentes'. De modo a elucidar o problema em questão, a Figura 30 ilustra o cenário hipotético. Neste exemplo são apresentados dois *Attributes* distintos, um do tipo *Boolean* (*atStatus*) e outro do tipo Double (*atTemperature*) em uma *Condition/Rule* composta por duas *Premises*. Uma *Premise* avaliaria se o estado de *atStatus* é verdadeiro, enquanto a outra *Premise* avaliaria se o estado de *atTemperature* é maior do que um dado valor.

Figura 30: Impacto nas alterações de estado de *Attributes* ativos

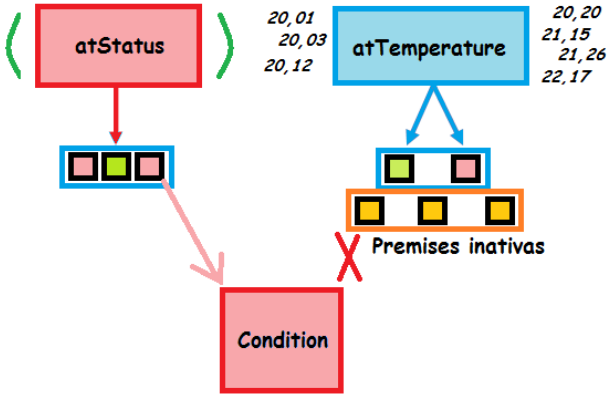


Fonte: Ronszcka, 2012

O *Attribute atStatus* apresentaria poucas mudanças em seu estado, permanecendo a maior parte do tempo com o estado *false*, disparando o fluxo de notificações esporadicamente. O *Attribute atTemperature*, por sua vez, apresentaria alterações constantes em seu estado, que no cenário em questão raramente impactariam na aprovação de sua *Condition* (RONSZCKA, 2012).

Neste sentido, um *Attribute* como *atTemperature* poderia ser categorizado como 'impertinente'. Isto dito, de modo a evitar tal cenário de notificações inúteis, cada *Attribute* impertinente em dado contexto deveria ter suas funções reativas desabilitadas temporariamente para com as *Premises-Conditions-Rules* afetadas pela impertinência. Neste âmbito, a Figura 31 considera o mesmo cenário anterior, agora com a inativação temporária das *Premises* contendo *Attributes* 'impertinentes' (RONSZCKA, 2012).

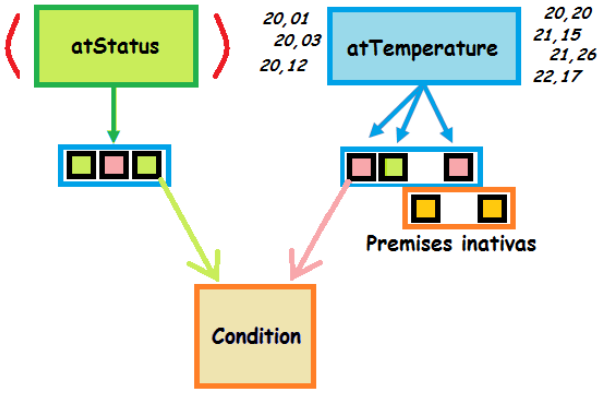
Figura 31: Impacto nas alterações de estado de *Attributes* 'impertinentes'



Fonte: Ronszcka, 2012

Conforme apresenta o cenário ilustrado na Figura 31, ao inativar temporariamente algumas das *Premises* compostas pelo *Attribute* impertinente *atTemperature*, as variações de estado desse não impactariam no disparo do fluxo de notificações para tais entidades. Neste âmbito, quando o conjunto dos *Attributes* tivesse aprovado 'suas' *Premises* em uma dada *Condition-Rule*, essa deveria solicitar a reativação das notificações para a *Premise* composta pelo *Attribute* impertinente. Assim, uma vez que o *Attribute atStatus* apresentasse estado verdadeiro, a *Condition-Rule* ilustrada solicitaria a reativação da *Premise* correspondente ao *Attribute atTemperature*, conforme ilustra a Figura 32.

Figura 32: Exemplo de reativação de uma entidade temporariamente desativada



Fonte: Ronszcka, 2012

Nesse sentido, conforme o cenário ilustrado na Figura 32, o *Attribute atStatus*, ao apresentar o estado verdadeiro, colocaria tal *Condition* em “ponto de aprovação”, o que reativaria as funções reativas da *Premise* temporariamente desconsiderada. Assim, a entidade *Premise* em questão permitiria ser notificada (e mesmo demandaria notificações) novamente pelo *Attribute atTemperature*, atuando normalmente como uma entidade (re)ativa. Por fim, após a devida aprovação e execução da *Rule* em questão, o *Attribute* impertinente voltaria a ignorar tal *Premise* até que seja requisitado novamente por outra *Condition* (RONSZCKA, 2012).

De maneira geral, em termos práticos, esse recurso é viabilizado no F*ramework* PON C++ 2.0 por intermédio da implementação do próprio padrão de projeto *Observer*. De acordo com a apresentação desse padrão no âmbito das entidades PON, essas já apresentavam mecanismos para adicionar e remover entidades interessadas em mudanças em seu estado. Assim, entidades passíveis de impertinência atualmente fazem uso desses mecanismos em tempo de execução (RONSZCKA, 2012).

3.6.7 Agregações de *Rules* em *FBEs – FBE Rules*

De maneira geral, as agregações de *Rules* em *FBEs* é um conceito próximo ao das *Formation Rules*. Tal conceito foi chamado anteriormente de *FBE Rules* e foi implementado inicialmente no Framework PON C++ 2.0 (RONSZCKA, 2012). Em suma, as *FBE Rules* consistem em um caso particular de *Formation Rules*, na qual a *Rule* está relacionada apenas a um determinado tipo de *FBE*, enquanto nas *Formation Rules* uma *Rule* pode estar relacionada a mais de um *FBE* (SANTOS, 2017).

Em outras palavras, a *Formation Rule* aparece como um elemento com escopo global na aplicação, criando relacionamentos *N* para *M* entre os *FBEs* participantes dessa regra especial. Os *FBE Rules*, por sua vez, possuem um escopo local ao *FBE* e todas as instâncias possuiriam obrigatoriamente uma instância da *Rule* em questão, formando um relacionamento *N* para 1.

A título de exemplo, considera-se um cenário hipotético de um sistema de alarme, no qual um dado tipo de sensor infravermelho precisaria atingir um certo nível de radiação para ativar a detecção de disparo. Para essa característica seria necessário implementar uma *Rule* específica para cada instância desse sensor. No caso de um cenário com um número elevado de sensores, sem a utilização do conceito de *FBE Rules* seria necessário replicar as *Rules* manualmente para cada instância declarada. Por outro lado, com a utilização do conceito de *FBE Rules* seria possível criar uma *Rule* genérica que seria instanciada automaticamente a cada novo sensor adicionado ao sistema.

3.6.8 Agregações entre *FBEs*

Outra característica pertinente ao PON é a possibilidade de criar agregações entre *FBEs*. Além dos tipos de dados primitivos (*i.e., boolean, char, integer, float* e *string*). Essa característica permite criar *FBEs* compostos, sem a necessidade de replicar múltiplos *Attributes*, um para cada característica de um Sensor, por exemplo, tornando o código mais coeso e sucinto.

Ademais, esse conceito tem como principal objetivo organizar a descrição de programas em PON, independente da forma de expressão (*i.e.,* gráfica ou textual), uma vez que as *Rules* poderiam ser compostas internamente aos *FBEs* (com base no conceito do conceito de *FBE Rules*) e igualmente com base em *FBEs* compostos. Com isso, espera-se atingir níveis de organização mais efetivos, melhorando particularmente a escrita e a legibilidade de programas baseados no PON.

A título de exemplo, considera-se um cenário hipotético de um sistema de alarme, no qual um conjunto de sirenes seriam vinculadas (*i.e.,* agregadas) a uma central de alarme. Para cada instância da *FBE Central* seria criado uma instância de cada uma das *FBEs* agregadas, bem como o relacionamento pontual entre elas.

3.7 CONSIDERAÇÕES SOBRE O PON

Como um paradigma emergente, relativamente novo, que vem evoluindo desde os trabalhos de Simão (2001; 2005), o PON é uma alternativa aos paradigmas de programação vigentes orientados a sequencialidade e pesquisas, nomeadamente PI e PD e seus subparadigmas. Com o intuito de resolver algumas das principais deficiências dos paradigmas de programação usuais e mesmo modelos de computação vigentes (*e.g.*, forte acoplamento e redundâncias estruturais e temporais), o PON apresenta uma nova forma de estruturar sistemas computacionais que se orienta no desacoplamento do processamento facto-execucional do processamento lógico-causal, ambos organizados por entidades com papeis muito específicos e todas elas colaborativas por meio de notificações precisas.

Em linhas gerais, com base na estrutura geral do PON, organizada por meio do modelo ímpar de entidades reativas e notificantes, em conjunto com os conceitos e funcionalidades do próprio PON, seria possível organizar a execução adequada do fluxo de notificações para atingir as propriedades elementares vislumbradas em sua teoria. Apesar disso, nenhuma materialização do PON, até então, havia contemplado efetivamente todas as propriedades elementares do paradigma. De modo a resumir a efetividade de cada materialização apresentada, a Tabela 4 ilustra as propriedades elementares contempladas nas principais materializações do PON.

Tabela 4: Propriedades elementares contempladas nas materializações do PON

Materialização / Propriedade	Software						Hardware			
	FW Prot. 2005	FW 1.0 2009	FW 2.0 2012	FW 3.0 2014	FWS Java C# 2016	FW 3.0 +PON IP 2016	PONHD Prot. 2011	CoPON Prot. 2012	NOCA 1.0 2015	PONHD 1.0 2018
Prog. Alto nível										
Paralelismo				✓	✓	✓	✓	✓	✓	✓
Distribuição						✓				
Desempenho							✓	✓	✓	✓

Fonte: Autoria Própria

Nesse âmbito, no tocante às formas de se utilizar o PON para a construção de sistemas, materializações em software e hardware foram produzidas. A primeira versão elaborada foi a de um *Fram.ework* prototipal, evoluindo posteriormente para versões 1.0, 2.0 e, mais recentemente, 3.0 (SIMÃO, 2005; SIMÃO; STADZISZ, 2008;

BANASZWESKI, 2009; RONSZCKA, 2012; VALENÇA, 2012; BELMONTE *et al*., 2016). Outros *Frameworks* (*i.e.,* Java e C#) e evoluções outras (*i.e.,* PON IP) possibilitaram também experimentar parcialmente questões como paralelização por meio de *threads* e distribuição por meio de *sockets* (HENZEN, 2015; OLIVEIRA, 2016; TALAU, 2016; BARRETO, 2016; BARRETO *et al*., 2018, OLIVEIRA *et al.,* 2018). Tais materializações possibilitaram a criação de aplicações PON e consequente validação dos conceitos relacionados a esse paradigma, porém o desempenho das aplicações não condizia com o esperado à luz do cálculo assintótico do PON, assim como também apresentaram abordagens preliminares em torno de paralelização e distribuição do processamento em PON.

Ainda, as materializações em hardware permitiram que se contornasse os obstáculos impostos por sistemas operacionais e outros componentes presentes nos computadores convencionais e sequenciais, baseados em Von Neumann. Nesse interim, por meio da utilização de VHDL e dispositivos FPGA, as materializações PON-HD, CoPON e NOCA/NOCASim proporcionaram a experimentação de aplicações realmente paralelas, demonstrando o conceito de desacoplamento implícito do PON (WITT *et al*., 2011; PETERS, 2012; LINHARES, 2015; PORDEUS, 2017; KERSCHEBAUMER, 2018).

Em suma, embora tenha havido avanços nas materializações tanto em software quanto em hardware, observou-se que tais materializações não proporcionam um resultado satisfatório por ferir uma ou outra das propriedades elementares do PON. As materializações em software via *frameworks* são capazes de prover alguma facilidade de programação e mesmo paralelismo em multinúcleo, mas não conseguem atingir um tempo de processamento adequado, devido a limitações da estrutura da ferramenta orientada a estruturas de dados dispendiosas computacionalmente. Por outro lado, as materializações em hardware apresentam tempo de processamento satisfatório e também paralelismo intrínseco, mas isso em detrimento da facilidade de programação, além de ainda não terem explorado a distribuição.

Um dos motivos para que nenhuma das materializações tenha atingido êxito em contemplar o PON em sua plenitude pode ter sido inclusive porque nenhuma delas havia implementado efetivamente todos os conceitos e funcionalidades do paradigma que ajudariam a alcançar as propriedades elementares, conforme discutido ao longo

deste presente capítulo. Nesse âmbito, a Tabela 5 apresenta uma síntese das principais materializações em relação aos conceitos implementados.

Tabela 5: Conceitos do PON contemplados nas materializações do paradigma

Materialização / Conceitos	Software						Hardware			
	FW Prot. 2005	FW 1.0 2009	FW 2.0 2012	FW 3.0 2014	FWS Java C# 2016	FW 3.0 +PON IP 2016	PONHD Prot. 2011	CoPON Prot. 2012	NOCA 1.0 2015	PONHD 1.0 2018
Reatividade das entidades	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Escalonamento de *Rules*		✓	✓	✓	✓	✓			✓	
Estratégias de resolução de conflito		✓	✓	✓	✓	✓			✓	
Compartilhamento de entidades			✓	✓	✓	✓				
Regras de formação										
Propriedades reativas dos *Attributes*			✓	✓		✓			✓	✓
Master Rule			✓	✓		✓				
Entidades impertinentes			✓	✓		✓				
FBE Rules			✓	✓		✓				
FBE agregador										

Fonte: Autoria Própria

Com base nos conceitos fundamentais do paradigma, o PON permite criar sistemas computacionais de forma natural, em alto nível, uma vez que possui inspiração na programação declarativa (*e.g.,* fatos e regras). Ainda, o inovador mecanismo de notificações do PON, permite a execução de sistemas computacionais de forma não redundante e minimamente acoplada (dita, portanto, desacoplada). Composto por pequenas entidades reativas, as quais colaboram por meio de notificações precisas e pontuais, o PON permite o paralelismo ou distribuição do processamento em um nível de granularidade mínimo.

Nesse sentido, o PON ainda carece de uma materialização efetiva que demonstre, de forma conjunta, as três propriedades elementares do paradigma que são que são (a) facilidades com o desenvolvimento em alto nível com base na estruturação organizada dos elementos do modelo, (b) baixo tempo de processamento pela ausência de redundâncias e (c) modelo preparado para uma execução paralela ou distribuída em função do desacoplamento implícito entre as

entidades. Assim, visando abstrair as características do PON em uma linguagem de alto nível e própria, bem como a concepção de aplicações com melhor desempenho à luz do cálculo assintótico, uma linguagem de programação e um compilador próprio foram vislumbrados como uma solução promissora para definir uma materialização efetiva para o PON.

Entretanto, observou-se logo nos primeiros protótipos dessa linguagem e compilador que a teoria de compilação tradicional, por si só, não apresentava as características desejadas e efetivamente adequadas para a criação de compiladores próprios para o PON. Isto se dá principalmente porque as técnicas e métodos de compilação tendem a ser especializadas nos paradigmas vigentes, orientados a pesquisa sob elementos passivos (*i.e.,* orientados a sequencialidade), principalmente quando mapeados em forma de árvores de estruturas algorítmicas.

Nesse âmbito, um método de compilação, apoiado na proposição de técnicas e mesmo conceitos adequados às características do PON (modelo orientado a entidades reativas e notificantes) se mostra desejável para a criação consistente de linguagens e compiladores para o Paradigma Orientado a Notificações (PON). Tal método é o objeto de estudo deste presente trabalho e é apresentado em maiores detalhes no próximo capítulo desta tese de doutorado.

CAPÍTULO 4

MÉTODO MCPON

Este capítulo apresenta em detalhes o método MCPON. Em suma, o MCPON é um método para a criação de linguagens e compiladores para o PON em plataformas distintas. Nesse âmbito, o capítulo está descrito em 4 seções. Primeiramente, a Seção 4.1 apresenta uma visão geral do MCPON, suas raízes e objetivos. A Seção 4.2, por sua vez, apresenta o Grafo PON como um balizador do método MCPON. A Seção 4.3, a seu turno, apresenta de maneira detalhada cada uma das etapas e subetapas do método MCPON. A Seção 4.4, por sua vez, apresenta um modelo estrutural de compiladores e sistemas de compilação baseados no método MCPON. Por fim, a Seção 4.5 apresenta as considerações finais do capítulo.

4.1 VISÃO GERAL DO MCPON

Conforme apresentado no Capítulo 2, mais especificamente na Seção 2.2.2, a teoria de construção de compiladores tradicional emprega métodos e técnicas de compilação cujo objetivo é basicamente mapear a estrutura dos programas em representações intermediárias. Geralmente, tais representações são mapeadas em forma de árvores (*e.g.,* árvore sintática, árvore de derivação etc.), mas também podem ser expressas por meio de instruções encadeadas em linguagens de baixo nível como código de três endereços ou linguagens outras que se baseiam em *SSA* (*Static Single Assignment*).

Em geral, tais métodos e técnicas são voltados para a compilação de linguagens regidas pelo Paradigma Imperativo (PI), incluindo nesse âmbito as linguagens regidas pelo Paradigmas Procedimental (PP) e pelo Paradigma Orientado a Objetos (POO). Tais métodos e técnicas também se aplicam, em partes, para o Paradigma Declarativo (PD), particularmente no âmbito do Paradigma Funcional (PF). No tocante ao Paradigma Lógico (PL), eles também se aplicam a alguns compiladores que traduzem código Prolog para C, ou até mesmo *SBRs* que são compilados para C/C++, conforme apresentado em detalhes no Capítulo 2.

De maneira geral, em todo caso, os métodos e técnicas utilizados na teoria de compilação tradicional não apresentam as características desejadas e

efetivamente adequadas para a criação de compiladores próprios para o PON. Isso porque o PON se baseia em uma nova forma de construir e conectar as colaborações de um programa por meio de suas entidades notificantes, conforme detalhado no Capítulo 3, mais especificamente na Seção 3.2. Em suma, a título de lembrança, entidades facto-execucionais chamadas de *FBEs* notificam pontualmente entidades lógico-causais chamadas de *Rules* por meio de um conjunto de sub-entidades que as organizam, reformulando a computação em relação ao que existia, cunhando assim um novo paradigma.

Nesse âmbito, este trabalho propõe um novo método para uniformizar o processo de construção de linguagens e compiladores específicos para o PON em plataformas distintas. A esse método foi dado o nome de MCPON (Método de Compilação para o PON). O primeiro passo para estruturação do MCPON foi a definição de um conjunto de diretrizes e regras para a construção de uma representação intermediária adequada para programas PON. Tal representação visa substituir apropriadamente as representações baseadas em árvore e, mesmo, as representações codificadas em linguagens de mais baixo nível, dos métodos de compilação tradicionais, os quais são pouco adequados ao PON.

Em resumo, o PON é constituído por um conjunto de entidades distintas e colaborativas que, ao propagarem suas notificações, dão fluência a uma dada lógica ou programa. Na prática, as ligações entre as entidades definem as colaborações por notificação de cada qual com as demais, colaborações essas que resultam nessa lógica conjunta. Nesse âmbito, vislumbrou-se que uma estrutura de dados em forma de grafo (e, portanto, não na forma de árvore ou codificações outras[9]) poderia representar apropriadamente um programa PON.

Para tal, cada programa escrito em PON teria uma representação única em forma de um grafo especializado, o qual representaria cada conexão por notificação entre cada uma das entidades desse programa PON. Sendo assim, este trabalho propõe essa nova estrutura geral de compilação para o MCPON que é denominada

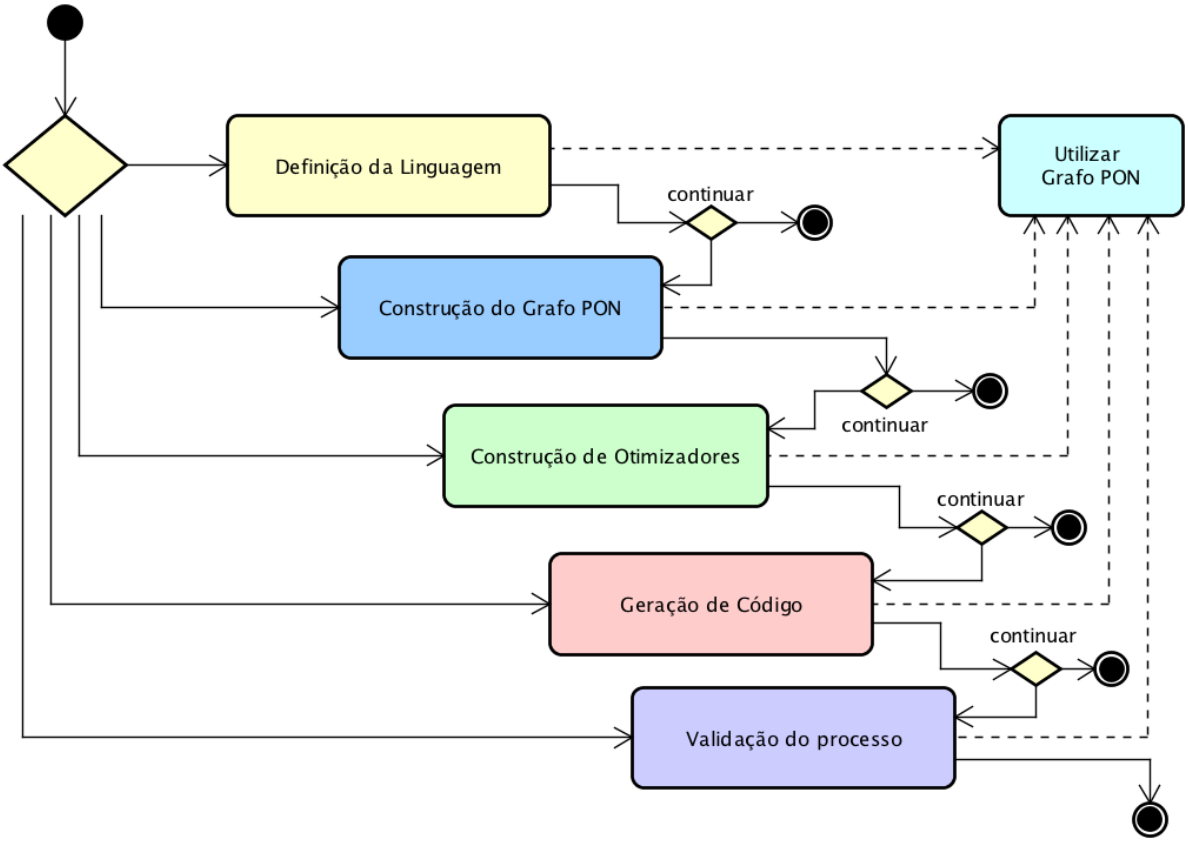
[9] Em geral, as linguagens intermediárias de mais baixo nível utilizadas no processo de compilação se orientam ao PI, compondo aninhamentos de instruções sequenciais e orientadas a pesquisa, o que inviabiliza as principais características do PON em relação aos seus princípios que se orienta a entidades notificantes não sequencializadas.

de Grafo PON. É importante ressaltar que cada programa em PON é percebido como uma instância de tal estrutura genérica.

Em suma, o advento do Grafo PON é tido como um elemento balizador especialmente desenvolvido para a criação de linguagens e compiladores particulares ao PON. Nesse sentido, vislumbrou-se que o Grafo PON serve como uma representação intermediária para o mapeamento completo de programas PON, e, principalmente, mantém a essência do PON, a qual é orientada a entidades notificantes desacopladas.

De maneira geral, o MCPON institui um sistema completo para o processo de compilação, sendo seus constituintes o já introduzido Grafo PON e cinco etapas que se especializam cada qual conforme seu papel. Nesse âmbito, a Figura 33 ilustra uma visão geral das etapas do método MCPON.

Figura 33: Visão geral das etapas do MCPON



Fonte: Autoria Própria

Conforme apresenta a Figura 33, o método MCPON é constituído por cinco etapas, as quais são fortemente dependentes do Grafo PON. Em linhas gerais, a primeira etapa do método visa essencialmente construir linguagens particulares para

o PON. A segunda etapa visa definir o processo de construção de instâncias do Grafo PON, inter-relacionando-se com a primeira etapa para este fim. A terceira etapa visa a construção de otimizadores, com o objetivo de eliminar possíveis redundâncias nos grafos gerados. A quarta etapa visa a transformação/tradução dos grafos em códigos-alvo tanto em linguagens quanto em plataformas distintas. Por fim, a quinta etapa visa a construção de validadores, com o objetivo de verificar a integralidade de cada compilador arquitetado.

Em linhas gerais, o MCPON proporciona, por meio do Grafo PON, o desacoplamento entre o *front-end* e o *back-end*, permitindo criar uma base uniforme para a construção de linguagens e compiladores para o PON. De fato, conforme salientado anteriormente, o desenvolvimento de linguagens e compiladores pode ser dividido em etapas, as quais podem se apresentar de forma desacoplada umas das outras, possibilitando a interação pontual e particular de cada etapa por parte dos desenvolvedores.

Ao seguir as diretrizes do método proposto neste trabalho, o desenvolvedor poderá criar sua própria linguagem para o PON, a qual se beneficiaria de uma base sólida (*i.e*., ferramentais construídos, testados e aplicados) para as etapas de compilação de sua linguagem. Além disso, com base em um grafo devidamente otimizado e livre de redundâncias é possível criar geradores de código especializados em construir programas em outras linguagens com uma estrutura de execução orientada a notificações (*i.e.,* aderente ao PON), de maneira a viabilizar a execução de tais programas em plataformas distintas. Por fim, conforme a última etapa do MCPON, o próprio Grafo PON pode auxiliar no processo de validação dos *targets* desenvolvidos, uma vez que os desenvolvedores podem mapear pequenos programas, com o intuito de validar a plenitude do *target* desenvolvido em relação aos conceitos do paradigma.

Em suma, este trabalho de pesquisa destaca-se por oferecer uma nova maneira de construir linguagens e compiladores para o PON que apresentam certa "independência de plataforma" (*i.e.,* não acoplamento de plataformas à priori) tanto no âmbito de linguagens quanto no âmbito de compiladores. É importante salientar que todas as etapas são integradas e interligadas, por meio do Grafo PON e, ao mesmo tempo, apresentam coesão e desacoplamento entre si, o que permite a evolução pontual de cada etapa sem afetar diretamente as demais etapas. Neste âmbito, a próxima subseção apresenta especificamente o Grafo PON e suas particularidades.

4.2 GRAFO PON

De maneira geral, a estrutura do PON, definida por meio das entidades notificantes que compõem seu modelo de execução e, portanto, influenciam sua programação, instigou o advento de uma inovação no processo de compilação para o PON. Tal inovação permite traduzir programas distintos escritos em linguagens próprias ao PON em uma única estrutura uniforme de dados. Tal estrutura possui um formato de grafo genérico, no qual é possível acomodar representações de todas as entidades extraídas e suas relações por notificação de um programa PON, as quais em conjunto formam a definição completa deste programa. Para essa estrutura genérica em forma de grafo foi dado o nome de Grafo PON.

Conforme explicitado na visão geral do método, o Grafo PON absolutamente se constitui no elemento central que rege a existência do método. Dito de outra forma, o Grafo PON é o elemento balizador que permite compor um método distinto de compilação para o PON, conformado à natureza deste novo paradigma. Nesse âmbito, esse método é diferente dos métodos tradicionais justamente por proporcionar as características necessárias para comportar as particularidades deste paradigma inovador, o que, naturalmente, leva o próprio método a ser igualmente inovador.

Diferente dos métodos tradicionais que mapeiam as características dos elementos de um programa em uma tabela de símbolos, bem como a estrutura sintática do programa em árvores de derivação e, posteriormente, em um conjunto de árvores sintáticas, ou representações intermediárias outras (muitas vezes, inclusive, com várias camadas de representações distintas), no MCPON, o mapeamento das entidades é feito unicamente no Grafo PON. Na verdade, a utilização daquelas árvores não se encaixam perfeitamente para o PON, uma vez que a construção de um programa PON não segue os princípios de programação sequencial orientada a pesquisa, mas sim, um modelo de interconexão de entidades notificantes, o que, de fato, motiva a criação de uma estrutura particular e própria como o Grafo PON.

De maneira geral, seguindo as diretrizes para a composição dos grafos especializados à luz do Grafo PON (*i.e.,* instâncias do Grafo PON), tais grafos armazenariam cada um dos elementos de um dado programa de forma particular, ainda que padronizada, permitindo mapear tanto as particularidades das entidades que compõe o programa, quanto as conexões de toda a cadeia de notificações. Essa característica particular do Grafo PON possibilita a reconstrução de qualquer

programa escrito sob os conceitos do PON em uma única representação intermediária especializada.

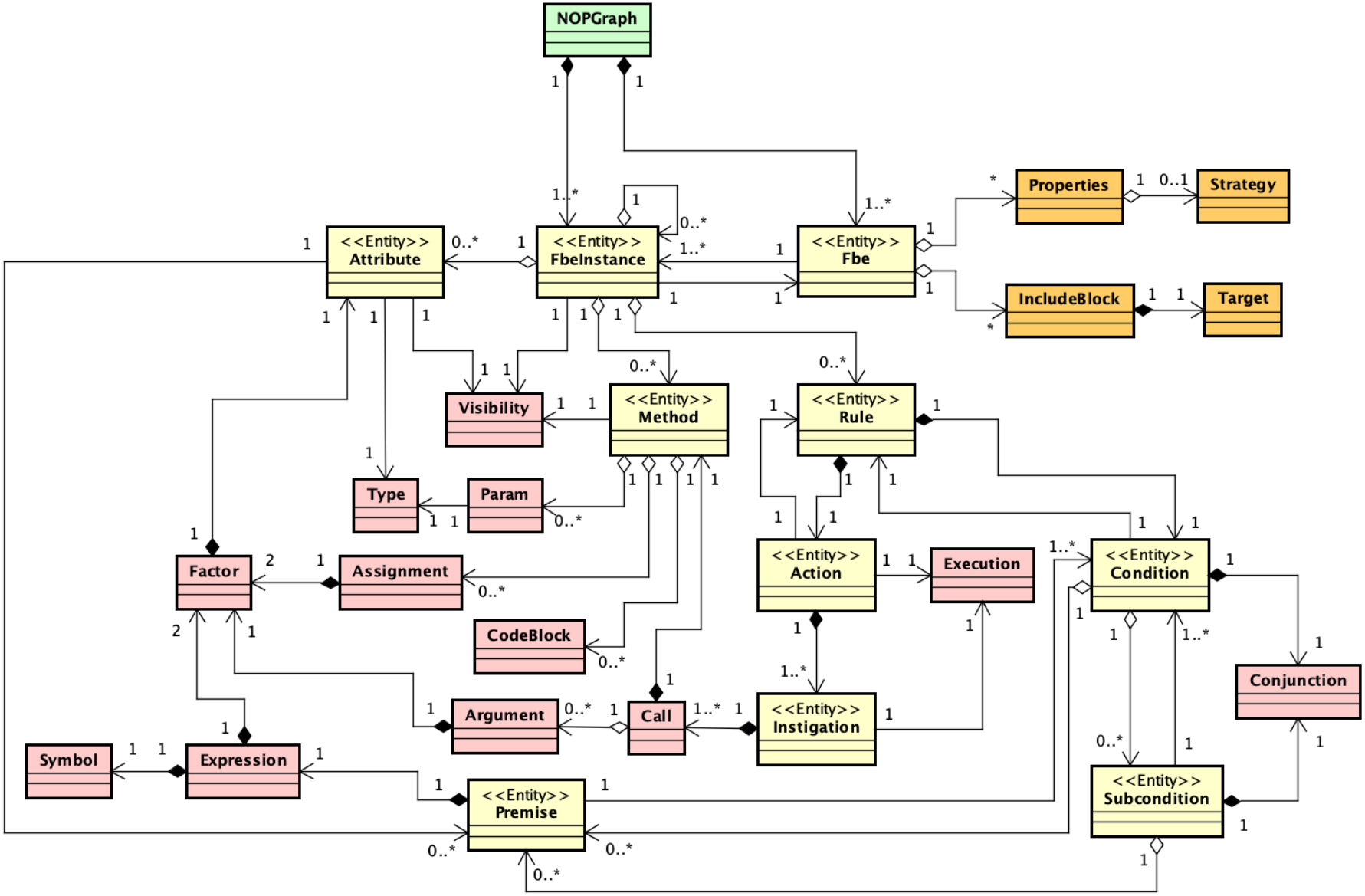
Em suma, cada instância do Grafo PON é uma representação completa de todas as entidades participantes do processo de inferência por notificações, criado especificamente para representar as ligações entre os elementos de um dado programa em PON. Na prática, todo e qualquer programa desenvolvido para o PON pode ter sua representação mapeada em instâncias do Grafo PON, desde que sejam seguidas algumas diretrizes para a construção de tal estrutura. Isto posto, o diagrama de classes ilustrado na Figura 34 apresenta o modelo genérico de entidades e suas dependências, bem como possíveis conexões.

De maneira prática, os elementos que compõem o Grafo PON são divididos em quatro grupos. O primeiro grupo, composto apenas pela entidade central do grafo (*i.e*., *NOPGraph*), representada no diagrama pela cor verde, define como as entidades principais são acomodadas em tal estrutura.

A entidade *NOPGraph* é o ponto central do mapeamento das entidades de um programa em PON. Ela possui relação **1:1..*** (*i.e*., um para um ou muitos) com as definições de *FBEs*, bem como relação de **1:1..*** com as instâncias de *FBE* (*FBEInstance*). De maneira geral, um Grafo PON deve conter pelo menos uma definição e uma instância de *FBE* para ser considerado um programa PON.

Ademais, o Grafo PON é composto, principalmente, por entidades que compõe o modelo original do PON. Tais entidades compõem essencialmente o grupo das entidades principais, representadas pela cor amarela, conforme ilustra o diagrama de classes em UML da Figura 34.

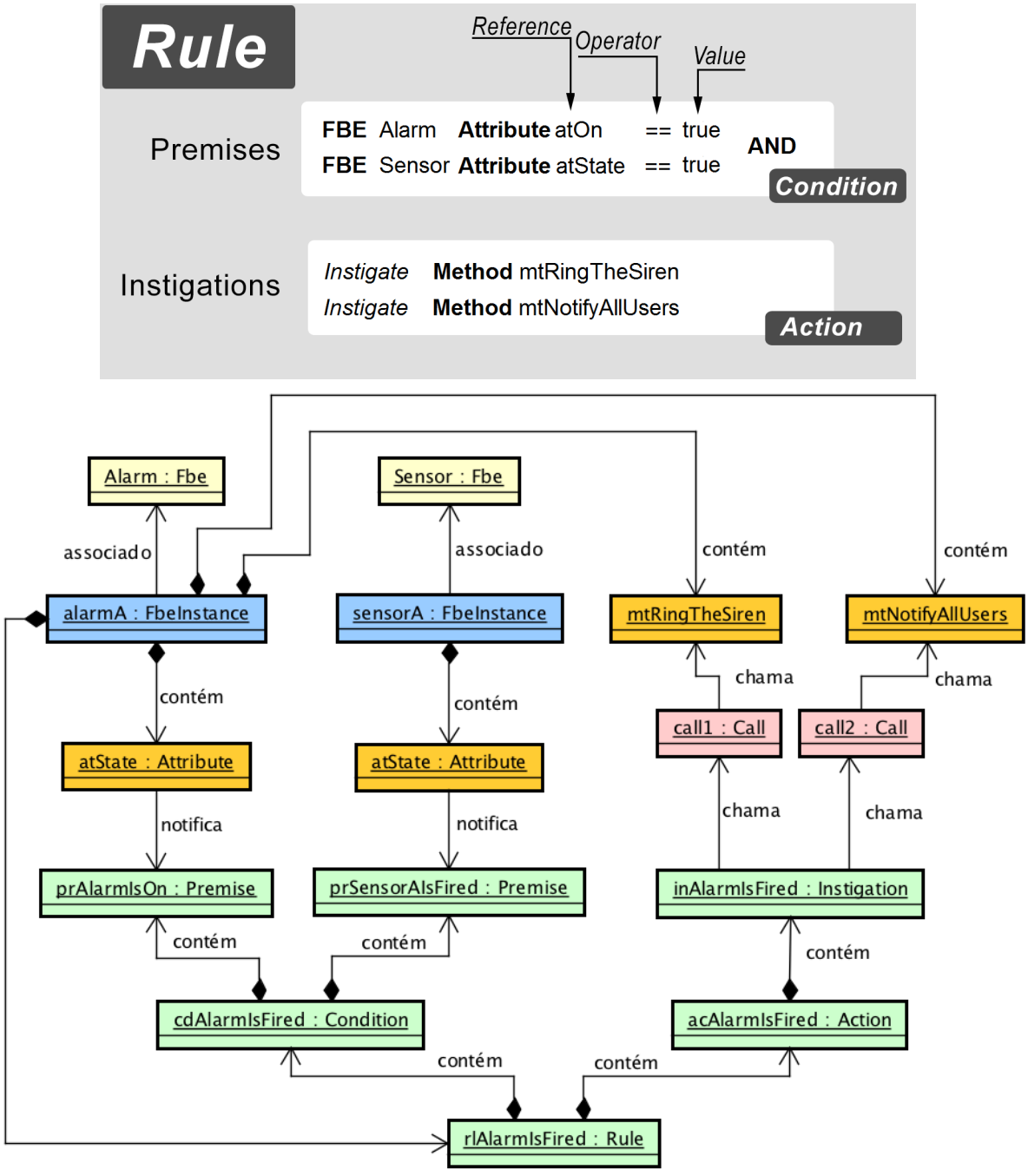
Figura 34: Representação do Grafo PON



Fonte: Autoria Própria via Diagrama de classes em UML

A título de elucidação, a Figura 35 apresenta uma *Rule* hipotética de um programa PON, bem como a respectiva instância do Grafo PON para representá-la. Tal grafo representa, por meio de seus constituintes, as entidades declaradas no programa exemplo apresentado. Quando se observa o grafo em questão, compreende-se que declarativamente está se definindo um conjunto de relações entre as entidades notificantes do PON que representam apropriadamente tal *Rule*.

Figura 35: Exemplo de programa mapeado em uma instância do Grafo PON



Fonte: Autoria Própria via Diagrama de objetos em UML

A seguir são detalhados os elementos que permitem representar cada programa em PON como uma instância do Grafo PON:

- ***FBE***: definição genérica de uma entidade *FBE*, a qual mapeia algumas particularidades compartilhadas por suas instâncias como propriedades e demais características particulares à etapa de geração de código. Tal entidade possui relação **1:1..*** com instâncias de *FBE*, o que implica na existência de pelo menos uma instância de *FBE* para cada *FBE* criada. Uma entidade *FBE* também pode possuir um conjunto de propriedades, representadas pela relação **1:*** (*i.e*., um para muitos) com *Properties* e uma relação **1:*** com *IncludeBlock*.

- ***FBEInstance***: instância de um *FBE*. Esta entidade é basicamente utilizada para representar cada instância particular de um *FBE*, mapeando o estado interno desta, bem como a conexão com a parte facto-execucional (*i.e., Attributes* e *Methods*) e com a parte lógico-causal, a qual é representada por *Rules* e demais entidades colaboradoras (*i.e., Condition, Premise, Action* e *Instigation*). Na prática, um *FBEInstance* possui uma associação com seu respectivo *FBE* "pai", acessando todas as propriedades genéricas deste. Além disso, instâncias de *FBE* podem agregar *N* outras instâncias de *FBE*, criando assim, uma relação hierárquica entre elas.

- ***Attribute***: entidade que representa cada *Attribute* particular de uma instância de *FBE*. Ela possui relação **1:1** (*i.e*., um para um) com *Visibility* e com *Type*, os quais definem basicamente a visibilidade de acesso (*i.e*., *public* e *private*) e o tipo desta (*i.e*., *Integer, Boolean, Float, Char* ou *String*), respectivamente. Ademais, tal entidade armazena seu valor inicial, bem como o nome deste no âmbito de cada programa compilado. Ainda, a entidade *Attribute* possui uma relação **1:0..*** com a entidade *Premise*, especialmente para determinar a relação por notificação entre elas, quando houver.

- ***Method***: entidade que representa cada ocorrência de definição de *Method* em instâncias de *FBE*. Assim como na definição de *Attributes*, os *Methods*

possuem uma relação **1:1** com *Visibility*. Em geral, um *Method* pode possuir representações que podem tanto ser simples atribuições de valores (*Assignment*), quanto a execução de um bloco de código particular à outras linguagens e paradigmas (*CodeBlock*). Em tempo, ele pode contar ou não, por meio de uma relação **1:0..***, com um conjunto de parâmetros (*Params*) para sua execução.

- ***Rule***: entidade que representa uma *Rule* particular à uma dada instância de *FBE*, recordando que tudo começa a partir de uma *FBE* principal. Basicamente, uma *Rule* é composta por uma *Condition* e por uma *Action*, ambas representadas por uma relação 1:1.

- ***Condition***: entidade que representa uma *Condition* particular à uma dada *Rule*. Tal entidade basicamente define a relação entre as entidades agregadas no tocante ao cálculo-lógico causal de uma *Rule*. Em geral, cada *Condition* possui uma relação **1:0..*** com a entidade *Premise*, assim como também com a entidade *Subcondition*. Em conjunto com as entidades colaboradoras está a definição do tipo de conjunção (ou disjunção) entre estas, definida por meio da entidade *Conjunction*.

- ***Subcondition***: entidade que representa uma *Subcondition* particular à uma dada *Condition*. Em geral, ela possui as mesmas características que uma *Condition* tradicional, com exceção de estar associada à uma *Condition* e não à uma *Rule*. Além disso, a *Subcondition* só apresenta relação **1:0..*** com a entidade *Premise* e não com outras *Subconditions*. Em conjunto com as entidades colaboradoras está a definição do tipo de conjunção (ou disjunção) entre estas, definida por meio da entidade *Conjunction*.

- ***Premise***: entidade que representa uma *Premise* na cadeia de notificações do PON. Para o Grafo PON, esta entidade armazena um conjunto de informações pontuais sobre a relação por notificação entre a *Premise* e seus respectivos *Attributes*. Entretanto, esta relação é composta por um elemento auxiliar denominado *Expression*, cuja relação é de **1:1** com

Premise, o qual define basicamente a expressão lógico-causal da *Premise* em questão.

- ***Action***: entidade que representa uma *Action* particular à uma dada *Rule*. Tal entidade basicamente define a parte executional de uma *Rule*, por meio da relação **1:1..*** (um para um ou muitos) com a entidade *Instigation*. Na prática, a entidade *Action* precisa de pelo menos uma *Instigation* para ser considerada correta no âmbito de um programa PON. Além disso, ela possui uma relação **1:1** com a entidade *Execution*, que basicamente define o modelo de execução das *Instigations* relacionadas.

- ***Instigation***: entidade que representa uma *Instigation* de uma *Action*. Tal entidade possui uma relação **1:1..*** com a entidade auxiliar *Call*, responsável pelas chamadas pontuais aos *Methods* associados. É importante observar que uma *Instigation* também possui uma relação **1:1** com a entidade *Execution*, definindo o mecanismo de execução das *Calls* relacionadas.

Ainda, existe outro conjunto de entidades auxiliares, representadas pela cor vermelha, as quais permitem a interconexão desse conjunto de entidades principais, mapeando particularmente as características sensíveis de cada qual. Além disso, tais entidades possuem o objetivo de complementar a estrutura do grafo com informações particulares e pontuais de cada uma das entidades que as agregam.

- ***Visibility***: entidade responsável por definir a visibilidade de uma entidade PON. Em suma, uma entidade pode possuir dois tipos de visibilidade, que são basicamente a pública e a privada. A definição de visibilidade pode variar de *target* para *target* (código-alvo para o qual o compilador será destinado). Em geral, é assumido que uma entidade com visibilidade pública é acessível em um escopo fora do qual ela se encontra (*i.e.*, acessível externamente), enquanto uma entidade com visibilidade privada é acessível apenas pelo escopo ao qual se encontra. Na prática, as entidades associadas à entidade *Visibility* são *FBEInstance*, *Attribute* e *Method*.

- ***Type***: entidade responsável por definir o tipo de um *Attribute*. Basicamente, os tipos que esta entidade pode assumir são: *Integer, Boolean, Float, String* e *Char*. Ademais, esta entidade possui uma relação **1:1** com a entidade *Attribute*, o que, na prática, determina que a tipagem desta seja imutável e, portanto, fortemente tipada.

- ***Param***: entidade responsável por definir um parâmetro de execução para uma entidade do tipo *Method*. Basicamente, como mencionado anteriormente, um *Method* pode possuir um conjunto de parâmetros, os quais possuem uma relação **1:1** com *Type*, de modo a definir justamente o tipo do parâmetro instanciado.

- ***Assignment***: entidade responsável por definir uma operação de atribuição. Em suma, esta entidade conecta dois fatores, representados pela entidade *Factor*, de modo que o primeiro fator (*i.e*., a esquerda) receberia o valor do segundo fator (*i.e*., a direita) no instante de execução de tal operação.

- ***Factor***: entidade responsável por encapsular o tipo e o valor de um dado *Attribute*. Tal entidade é utilizada em operações de atribuição, em expressões lógico-causais, e em argumentos de chamadas de métodos. Em geral, a entidade *Factor* pode estar associada (*i.e*., **1:1**) à um *Attribute*, de qualquer tipo, de um dado programa, bem como à uma constante de qualquer tipo e valor, definida no código-fonte do programa. Esta entidade possui um único propósito, apesar de ser utilizada em várias operações distintas, que é basicamente encapsular um *Attribute* ou um valor constante de qualquer um dos tipos de dados do PON.

- ***CodeBlock***: entidade responsável por encapsular um bloco de código, geralmente escrito em outras linguagens e paradigmas, o qual é normalmente transportado (*i.e*., copiado integralmente) para o código-alvo gerado. Normalmente, este trecho de código é anexado na execução de um *Method* no programa PON gerado, de modo a executar código ‘nativo’ na plataforma-alvo. Esta entidade basicamente possibilita a programação

multiparadigmas em programas PON, permitindo e facilitando a integração de programas legados com programas escritos em PON.

- ***Call***: entidade responsável por mapear as chamadas pontuais aos *Methods* associados. Tal entidade possui uma relação **1:1** com uma instância particular de uma entidade *Method*. Ademais, tal entidade possui uma relação **1:0..*** com a entidade *Argument*, possibilitando a passagem de argumentos (*i.e.*, parâmetros em tempo de execução) para a cada chamada pontual de *Method*.

- ***Argument***: entidade responsável por encapsular as características de um argumento de uma chamada de *Method*. Basicamente, tal entidade encapsula o nome de um determinado argumento, bem como a entidade *Attribute* com a qual se relaciona, a qual também se encontra encapsulada em um *Factor*. A relação entre *Argument* e *Factor* é de **1:1**, assim como a relação entre *Factor* e *Attribute*, preservando a coesão entre eles.

- ***Expression***: entidade responsável por encapsular a expressão lógico-causal de uma *Premise*. Tal relação é obrigatoriamente binária (*i.e.*, **1:2**) para com um par de entidades do tipo *Factor*. Ademais, tal entidade possui uma relação **1:1** com a entidade *Symbol*, a qual define o tipo de símbolo utilizado na comparação entre os dois fatores da expressão em questão.

- ***Symbol***: entidade responsável por definir o tipo de comparação de uma *Expression*. Basicamente os tipos são: igual a (*EQUAL*), diferente de (*NOT EQUAL*), menor que (*LESSER THAN*), maior que (*GREATER THAN*), menor ou igual a (*LESSER OR EQUAL*), maior ou igual a (*GREATER OR EQUAL*).

- ***Conjunction***: entidade responsável por encapsular o tipo de conjunção de uma *Condition* ou de uma *Subcondition*. Basicamente, os tipos são conjunção (*and*) e disjunção (*or*). Para o primeiro caso, todas as *Premises* ou *Subconditions* associadas, sem exceção, devem estar aprovadas para

a aprovação da entidade associada a ela. Por outro lado, no caso da disjunção, apenas uma, das *Premises* ou *Subconditions* associadas, precisa estar aprovada para a aprovação da entidade associada a tal *Conjunction*.

- ***Execution***: entidade responsável por definir o tipo de execução de uma *Action* ou de uma *Instigation*. Basicamente os tipos são: (a) *sequential*, no qual a execução das entidades associadas deve ser exclusivamente sequencial, ou seja, permitir a execução de somente uma entidade por vez, uma após a outra, em fila, e somente permitir a execução da outra após o término da anterior; e (b) *parallel*, no qual a execução deve ser particularmente paralela e simultânea, levando em consideração as particularidades e restrições da plataforma-alvo adotada na etapa de geração de código.

Além das entidades principais e das entidades auxiliares, o modelo possui entidades complementares, particulares a entidade *FBE*, que em geral se resumem a classes responsáveis por parametrizações no processo de compilação, representadas pela cor laranja, conforme ilustra o diagrama de classes em UML da Figura 35.

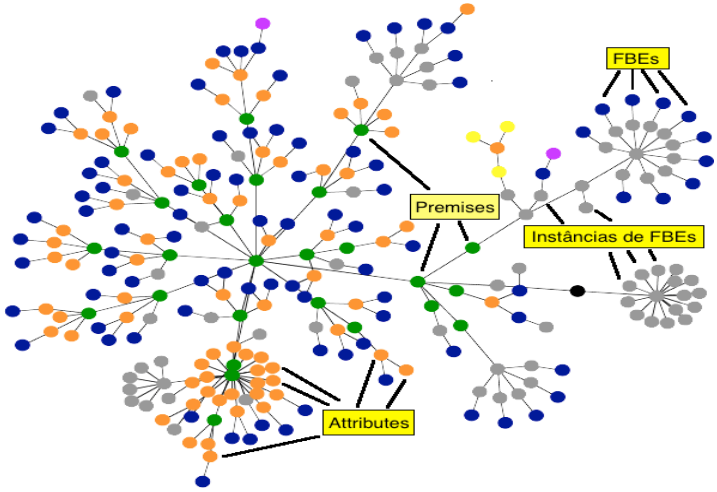
- ***Properties***: entidade responsável por definir as propriedades do processo de compilação de um *FBE*. Como uma opção de propriedade, tem-se a propriedade *Strategy*, para a definição da estratégia de escalonamento e de resolução de conflitos. Atualmente, esta entidade está aberta para adição de novas propriedades.
- ***Strategy***: entidade responsável por definir a estratégia de escalonamento e de resolução de conflitos das *Rules* internas à um determinado *FBE.* Como opções de estratégias tem-se: (a) *BREADTH* (escalonamento do tipo fila, *i.e.*, *FIFO*); (b) *DEPTH* (escalonamento do tipo pilha, *i.e.*, *LIFO*); e (c) *PRIORITY* (escalonamento baseado na prioridade das *Rules*). É importante ressaltar que cada uma das estratégias definidas precisa ter suas respectivas implementações contempladas nos *target*s no processo de geração de código.

- ***IncludeBlock***: entidade responsável por encapsular um bloco de inclusões e definições (*e.g.*, *includes* e *defines* do C/C++ ou *imports* do Java), com o objetivo de facilitar a integração do programa PON com o código nativo da plataforma-alvo. Tal bloco de inclusões normalmente é transportado (*i.e.*, copiado integralmente) para o código-alvo gerado.

- ***Target***: entidade responsável por definir o *target* para o qual a entidade *IncludeBlock* está associada.

É importante ressaltar que o diagrama de classes expõe as classes fundamentais da estrutura, as quais devem ser mapeadas pontualmente e individualmente, de acordo com a estrutura do programa a ser representado. Nesse sentido, de modo a elucidar o entendimento do Grafo PON, a Figura 36 apresenta uma representação análoga do que seria a estrutura de um programa qualquer.

Figura 36: Representação ilustrativa de um de Grafo PON



Fonte: Autoria Própria

A Figura 36 apresenta de maneira ilustrativa como seria a composição de uma instância do Grafo PON. Basicamente, cada recorrência de um elemento significativo em um dado programa (*e.g.*, *FBE,* Instâncias de *FBE, Attributes* de instâncias de *FBE* etc.) é representado por um nó no Grafo PON. Ademais, cada relação notificante entre uma entidade e outra é mapeada com ligações (*i.e.*, arestas) no grafo. Desta maneira, qualquer programa escrito com os princípios do PON pode ser mapeado de forma completa em uma única estrutura de dados.

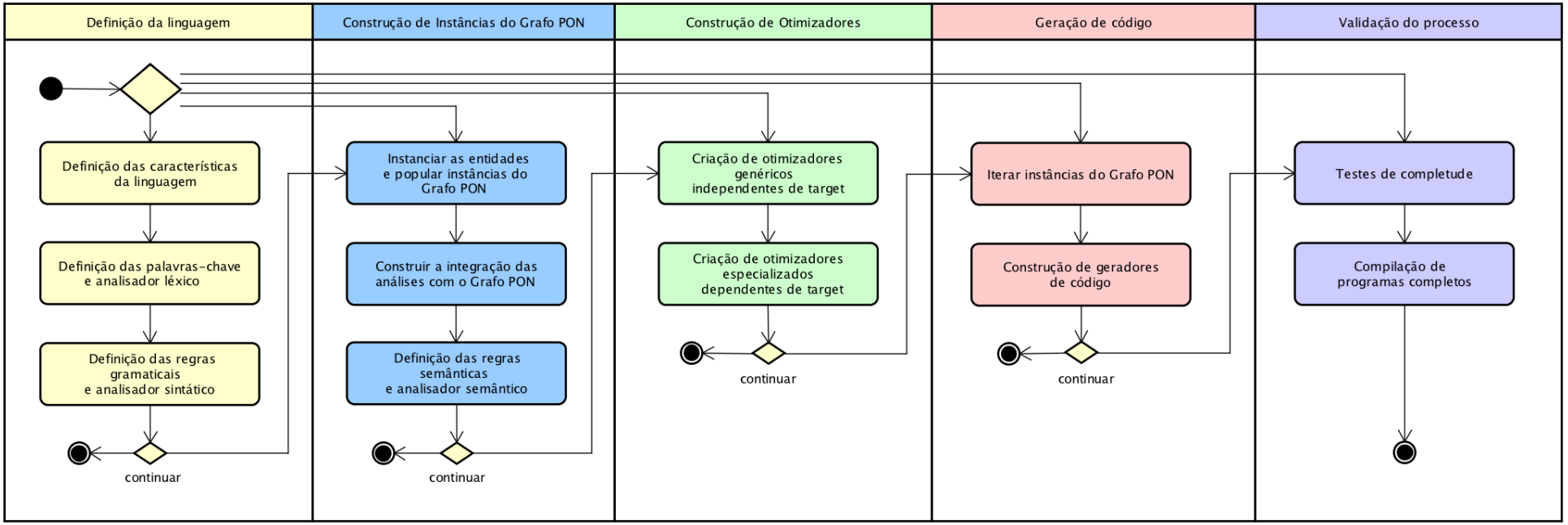
Em suma, a construção do Grafo PON tem por objetivo auxiliar no processo de compilação, permitindo mapear de forma completa a essência de cada programa em PON em uma dada instância do grafo. Com base neste mapeamento fidedigno é possível, posteriormente, realizar a tradução dessa instância do grafo para um código-alvo desejado. De fato, este processo constitui em uma das principais etapas de um compilador para o PON e, justamente por este motivo, precisou do advento do chamado Grafo PON para possibilitar um processo de compilação efetivo para o PON.

4.3 ETAPAS PARA A CONSTRUÇÃO DO MCPON

Conforme apresentado sumariamente na Seção 4.1, o MCPON é constituído por cinco etapas, as quais se completam com o objetivo principal de permitir construir linguagens e compiladores para o PON em plataformas distintas. Ademais, é importante ressaltar que o desenvolvedor pode utilizar qualquer ferramenta, técnica ou linguagem de programação para a implementação do método MCPON, com a ressalva de que o Grafo PON deve ser construído de acordo com as diretrizes definidas para sua composição.

Em suma, conforme ilustrado na Figura 33 e explicado na Seção 4.1, o MCPON apresenta as seguintes etapas: 1. Definição da linguagem; 2. Construção do Grafo PON; 3. Construção de Otimizadores; 4. Geração de código e 5. Validação do processo. Ademais, a Figura 37 ilustra em maiores detalhes as etapas e subetapas do MCPON.

Figura 37: Etapas e Subetapas do MCPON



Fonte: Autoria Própria via Diagrama de atividades em UML

Com base no ilustrado e modelado na Figura 37, explica-se agora o diagrama de atividades à luz das cinco principais etapas do MCPON e suas subetapas. De maneira geral, a Etapa 1 consiste na definição de uma linguagem especializada para o PON, o que inclui toda a elaboração desta, como suas particularidades e características (*e.g.*, gramática, estrutura sintática etc.). De fato, é nesta etapa que são definidas as palavras-chave utilizadas na definição dos elementos de um programa, bem como as regras gramaticais da linguagem para a respectiva construção dos analisadores léxico e sintático. O resultado gerado ao fim desta primeira etapa é essencialmente um programa que realiza toda a análise léxica e sintática desta nova linguagem, cujo mapeamento subsequente é integralmente direcionado para a construção de grafos especializados com base nas diretrizes do Grafo PON (Etapa 2).

Na verdade, o desenvolvedor poderia construir este programa de modo a integrar a Etapa 1 com a Etapa 2, onde a construção dos grafos especializados seria realizada durante o processo de análise de um dado programa. De maneira geral, a Etapa 2 consiste em instanciar os elementos do Grafo PON de modo a popular um grafo especializado, o qual representaria integralmente o programa definido como entrada na primeira etapa. O artefato gerado no fim da Etapa 2 é um programa que gera grafos especializados de acordo com as diretrizes do Grafo PON.

A Etapa 3, por sua vez, tem como objetivo principal auxiliar na construção de otimizadores para os grafos gerados, de modo a reduzir ou quiçá eliminar redundâncias presentes na estrutura de tais grafos, oriundos das etapas anteriores, assim como gerar otimizações específicas para os *targets* visados na Etapa 4 (*i.e.,* geração de código). Para isso, a Etapa 3 estabelece duas categorias de otimizadores. De um lado, do *front-end*, constitui-se os otimizadores genéricos, independentes de *target*. Do outro lado, do *back-end*, constitui-se os otimizadores especializados, dependentes do *target*.

A Etapa 4, particularmente, consiste na fase de síntese do processo de compilação. Em geral, para esta etapa são definidas as particularidades do *target* para o qual a compilação (*i.e.*, tradução/transcrição) dos grafos será destinada. Para isso, primeiramente, o desenvolvedor precisará definir a plataforma e linguagem-alvo para a qual se destina o *target*. Com base nisso, ele deve explorar as características do PON como a paralelização e/ou a distribuição tanto quanto a plataforma/linguagem-alvo permitir. Ademais, uma vez definidas a plataforma e a linguagem-alvo, o

desenvolvedor deverá construir um gerador de código apropriado para a tradução dos elementos das instâncias do Grafo PON em código-alvo na linguagem escolhida.

Por fim, a Etapa 5 consiste na validação do processo de compilação como um todo. Tal etapa, basicamente, contempla as validações dos *targets* gerados por meio de um conjunto de pequenos programas PON, em forma de grafos previamente criados (*i.e.*, programas interpretados e mapeados em função do Grafo PON). Tais instâncias do Grafo PON possuem o intuito de validar as principais características do PON por meio de testes minimalistas e pontuais que visam medir a integralidade e abrangência de conceitos implementados no compilador (*i.e.*, gerador de código) criado. Ainda, como complemento aos testes de validação, sugere-se a compilação de programas completos, os quais também podem estar mapeados na forma de instâncias de grafos completos.

De modo a apresentar o MCPON em maiores detalhes, as próximas subseções apresentam pontualmente cada uma das etapas e subetapas do método.

4.3.1 Definição da linguagem

A primeira etapa na construção de uma linguagem e compilador baseado no MCPON é a definição de uma linguagem de programação especializada para o PON. Nesse âmbito, esta etapa contará com três subetapas fundamentais que são: (a) definir as características e particularidades da linguagem a ser arquitetada; (b) definir as palavras-chave desta nova linguagem e, por fim, (c) definir a estrutura sintática da linguagem por meio de regras gramaticais.

Em suma, tais definições permitem que o desenvolvedor possa construir parcialmente o *front-end* de seu compilador, com base nas análises léxica e sintática. Com isso, o compilador terá a capacidade de interpretar um código-fonte, com base nas palavras-chave, identificadores e demais instruções criadas para tal linguagem, bem como as regras gramaticais especificadas para a mesma.

Ao concluir esta etapa, tem-se uma nova especificação de linguagem para o PON, bem como todo um ferramental para a construção inicial do *front-end* de um compilador específico para o PON. Nesse âmbito, de modo a nortear o desenvolvimento desta etapa, as próximas subseções explicam as subetapas em detalhes.

4.3.1.1 Definição das características da linguagem

Em geral, as linguagens de programação voltadas ao PON possuem um grau de abstração considerado de alto nível, composta por símbolos complexos que, apesar de serem inteligíveis pelos desenvolvedores, não são executáveis diretamente pela máquina. Isso se dá, principalmente, por tais linguagens serem de natureza declarativa e, portanto, fortemente dependentes de uma reconstrução elaborada por parte dos compiladores na transformação de código-fonte em código de máquina. Ademais, outras características que as linguagens devem levar em consideração foram apresentadas no Apêndice A, mais especificamente na Seção A.2.

Em geral, conforme salientado na Seção A.2.1, as linguagens de programação devem apresentar facilidades de escrita, com um conjunto simplificado de instruções, projetadas de maneira a proporcionar clareza de escrita. Isso diminui a curva de aprendizado de novos adeptos a linguagem, bem como facilita o processo de manutenção de software.

Nesse âmbito, o desenvolvedor deve levar em consideração que ao elaborar linguagens baseadas no PON, algumas das particularidades intrínsecas do paradigma podem ser preservadas, como a orientação a regras, por exemplo. É possível, porém, levar a linguagem para um nível mais detalhado, talvez de mais baixo nível, na qual a escrita de programas poderia identificar cada uma das entidades notificantes do modelo do PON e possibilitar que o desenvolvedor tenha mais detalhamento na programação como um todo. Como exemplo dessa última abordagem, a linguagem poderia permitir o direcionamento manual de execução das entidades para núcleos de processamento específicos, bem como nós de processamento distribuídos ou, até mesmo, permitir a programação multiparadigmas, permitindo o “enxerto” de outras técnicas de programação, viabilizando um certo hibridismo de programação.

Independentemente do caminho escolhido, a linguagem deveria levar em consideração a legibilidade como principal característica. Para isso, o Código 11 apresenta um exemplo de um programa que modela um sistema de alarme monitorado, em uma linguagem conformada ao PON.

Código 11: Exemplo de programa PON para um alarme monitorado

```
1  fbe Alarm
2    attributes
3      atState = false
4    end_attributes
5    methods
6      mtFireAlarm [atState = true]
7    end_methods
8  end_fbe
9
10 fbe Sensor
11   attributes
12     atState = false
13   end_attributes
14 end_fbe
15
16 fbe Siren
17   attributes
18     atFired = false
19   end_attributes
20   methods
21     mtRingSiren [atFired = true]
22   end_methods
23 end_fbe
24
25 inst
26   Alarm alarmA
27   Sensor sensorA, sensorB
28   Siren sirenA
29 end_inst
30
31 rule rlAlarmIsFired
32   condition cdAlarmIsFired
33     premise prSensorAIsFired [sensorA.atState == true]
34     or
35     premise prSensorBIsFired [sensorB.atState == true]
36   end_condition
37   action acAlarmIsFired
38     instigation
39        call alarmA.mtFireAlarm
40        call sirenA.mtRingSiren
41     end_instigation
42   end_action
43 end_rule
```

Fonte: Autoria Própria

Em linhas gerais, conforme apresenta o Código 11, o programa apresenta a construção das definições de três *FBEs* (linhas 1 a 23), assim como cria instâncias específicas destas entidades (linhas 25 a 29). Por fim, uma *Rule* define a relação entre todas as entidades do modelo (linhas 31 a 43).

Conforme é possível observar neste exemplo de programa, a expressão dos elementos do modelo é definida de forma clara e explícita, de modo a apresentar uma essência particularmente didática. Nesse âmbito, tal programa se apresenta de

maneira inteligível, possibilitando que mesmo desenvolvedores com pouca experiência em PON compreendam a estrutura lógica deste programa.

Ademais, por definição, as linguagens que são regidas pelo MCPON são fortemente e estaticamente tipadas, mesmo que na definição da linguagem não seja explicitado o tipo dos *Attributes*, conforme o exemplo do Código 11. Isto é, o tipo dos *Attributes* (*e.g.*, *Integer, Boolean, Float, String* e *Char*) são definidos em tempo de compilação, principalmente na etapa de constituição do Grafo PON, sendo que qualquer atribuição de valores incompatíveis deveria gerar erros de compilação. Além disso, os tipos dos *Attributes* também não se alteram durante a execução de um programa, conforme características apresentadas na Seção A.2.2.

De maneira geral, as linguagens a serem desenvolvidas com base no MCPON já apresentam um certo nível de confiabilidade, conforme Seção A.2.3, uma vez que se baseiam em um mecanismo de execução orientado a notificações. Em suma, o mecanismo de execução do PON poderia garantir um certo nível de determinismo de execução, mesmo em plataformas distintas, desde que suas entidades sejam implementadas corretamente e, em especial, os mecanismos de escalonamento de *Rules* e resolução de conflito sejam implementados apropriadamente.

Por fim, as linguagens devem apresentar a característica de ‘implementação eficiente’, conforme Seção A.2.7. Esta característica visa que o desenvolvimento de programas em uma linguagem seja uma atividade prática e intuitiva. Além disso, tal atividade deve ser simplificada, ao passo de tornar a leitura dos programas inteligível e, ao mesmo tempo, facilitar a escrita dos mesmos. Por “facilitar”, entende-se que a linguagem deveria conter um conjunto relativamente simplificado de *tokens*, bem como regras sintáticas coerentes e de fácil entendimento. Tudo isso contribui para a curva de aprendizado, para a velocidade de desenvolvimento, para a minimização de manutenções de programas legados, entre outros benefícios.

4.3.1.2 Definição das palavras-chave e analisador léxico

Com base na teoria das linguagens formais apresentadas na Seção 2.2.1, em especial, as linguagens regulares, conforme a hierarquia de Chomsky, a próxima subetapa para a criação de uma linguagem baseada no MCPON constitui-se na definição de um conjunto de palavras-chave. Além disso, com base em uma linguagem regular, é possível construir o analisador léxico desta linguagem, levando

em conta tais palavras-chave, bem como possíveis identificadores e outros símbolos pertinentes.

Tal conjunto de palavras-chave, bem como identificadores e instruções outras, são compostas de acordo com as expressões regulares definidas e constituem-se nos possíveis elementos de um programa. Em geral, o número de elementos e regras de construção léxica afeta a curva de aprendizado de uma linguagem de programação. Se ela se apresentar de maneira sucinta, com poucas variações e exceções, espera-se que menos tempo seja despendido para a compreensão desta linguagem. Por outro lado, é importante observar que linguagens que utilizam poucas palavras-chave, porém com um conjunto composto por muitos símbolos para representar diferentes variações de uma mesma expressão, podem igualmente comprometer a simplicidade da linguagem.

Exemplos de palavras-chave levantados no programa exemplo apresentado anteriormente, mais especificamente no Código 11, sugere as seguintes palavras-chave para esta linguagem hipotética: *fbe, end_fbe, attributes, end_attributes, methods, end_methods, true, false, inst, end_inst, rule, end_rule, condition, end_condition, action, end_action, premise, and, or, instigation, end_instigation* e *call*. É importante salientar que tais palavras-chave descrevem de maneira expressiva as principais entidades do modelo do PON, tornando a linguagem particularmente didática nesse sentido.

A definição das palavras-chave tem um papel fundamental no processo de compilação, mais especificamente na análise léxica, a qual tem como principal objetivo produzir uma sequência de *tokens* a partir dos lexemas encontrados durante a análise do código-fonte de um programa. Usualmente, nos analisadores léxicos, as palavras-chave são armazenadas em uma tabela interna a qual é verificada a cada lexema encontrado de modo a classificar este como uma palavra reservada ou como um identificador.

Em geral, um analisador léxico pode ser implementado pelo desenvolvedor por meio de linguagens regulares, autômatos finitos, expressões regulares ou gramáticas regulares, conforme considerado na Seção 2.2.1. Embora o desenvolvedor possa implementar seu próprio algoritmo para a identificação dos lexemas, existem diversas implementações para auxiliar no processo de criação de analisadores léxicos para diferentes linguagens de programação, tais como o Flex, JFlex, Turbo Pascal Lex/Yacc, Flex++ e CSLex, conforme (FLEX, 2019).

Em todo o caso, a notação para utilização destas ferramentas é conhecida como linguagem lex. Neste âmbito, o Código 12 apresenta um exemplo de especificação em linguagem lex, com base na ferramenta Flex, da linguagem utilizada para a apresentação do MCPON.

Código 12: Exemplo de regras léxicas para a LingPON

```
1  fbe                          return FBE;
2  method                       return METHOD;
3  rule                         return RULE;
4
5  [-+]?[0-9]+                  return INTEGER_VALUE;
6
7  [-+]?[0-9]*\.?[0-9]+         return DOUBLE_VALUE;
8
9  \'.\'                        return CHAR_VALUE;
10
11 \".*\"                       return STRING_VALUE;
12
13 [a-zA-Z\_]([a-zA-Z0-9\_])*   return ID;
```

Fonte: Autoria Própria

Em linhas gerais, as regras léxicas, apresentadas no Código 12, inicialmente verificam se os lexemas de entrada são classificados como as palavras-chave definidas. Caso contrário, são confrontadas com as regras para identificação de números, caracteres, cadeias de caracteres, identificadores etc. A saída do analisador léxico é uma cadeia de *tokens* que é passada para a próxima fase, a análise sintática. Geralmente, os analisadores léxicos são implementados como uma sub-rotina que funciona sob o comando do analisador sintático.

4.3.1.3 Definição das regras gramaticais e analisador sintático

Ainda, com base na teoria das linguagens formais apresentadas na Seção 2.2.1, em especial, as linguagens livres de contexto, conforme a hierarquia de Chomsky, a próxima subetapa para a criação de uma linguagem baseada no MCPON é a construção de um analisador sintático. Um analisador sintático é basicamente um algoritmo que faz uso dos *tokens* identificados pelo analisador léxico, comparando estes com um conjunto de regras gramaticais, com o objetivo de validar se a estrutura de um programa está de acordo com a gramática de uma determinada linguagem de programação.

Para a construção de um analisador sintático, o desenvolvedor poderia utilizar linguagens livres de contexto tradicionais, como a própria BNF, conforme apresentado em maiores detalhes na Seção 2.2.1. Atualmente existem vários programas que facilitam a criação de analisadores sintáticos. Na prática, tais programas são geradores de analisadores sintáticos, os quais recebem como entrada a gramática de uma linguagem e como saída proveem um algoritmo para a análise sintática desta linguagem. A título de exemplo, programas como Yacc, Bison, SableCC, COCO/R e AntLR são algumas das opções para a geração de analisadores sintáticos, conforme (BISON, 2019).

Neste âmbito, o Código 13 apresenta um exemplo de regras gramaticais (com a sintaxe do programa Bison, uma adaptação da própria BNF) para a linguagem utilizada para a apresentação do método MCPON.

Código 13: Exemplo de regras gramaticais para uma linguagem para o PON

```
1  PROGRAM              : fbes instances rules
2
3  fbes                 : fbe fbes
4                       | fbe
5
6  fbe                  : FBE ID attributes methods
7
8  attributes           : ATTRIBUTES attributes_body END_ATTRIBUTES
9
10 attributes_body      : attribute attributes_body
11                      | attribute
12
13 attribute            : ID EQUAL VALUE
14
15 methods              : METHODS methods_body END_METHODS
16
17 methods_body         : method methods_body
18                      | method
19
20 method               : ID OPENBR ID EQUAL VALUE CLOSEBR
```

Fonte: Autoria Própria

Em linhas gerais, as regras gramaticais apresentadas no Código 13, contemplam parcialmente a linguagem hipotética criada para exemplificação do MCPON. Basicamente, a linha 1 determina que um programa precisa conter um conjunto de *FBEs*, um conjunto de instâncias de *FBE* e um conjunto de *Rules*. O conjunto de *FBEs* pode conter um ou mais *FBEs* (linhas 3 e 4) que são representados, na prática, pela palavra-chave *FBE*, por um identificador e um conjunto de *Attributes*

e *Methods* (linha 6). Ademais, o restante do código segue a mesma lógica, definindo o que cada grupo representa na estrutura gramatical desta linguagem.

Ainda, a subetapa de análise sintática normalmente é integrada à construção de uma estrutura de dados que acomodaria os elementos extraídos do código-fonte no processo de compilação. Nesse âmbito, seguindo as diretrizes do método MCPON, o desenvolvedor deve instanciar o Grafo PON e popular esta instância, de modo a contemplar esta integração. Para isso, a próxima etapa apresenta, em detalhes, as regras para a construção de instâncias do Grafo PON.

4.3.2 Construção de Instâncias do Grafo PON

Em suma, conforme considerado anteriormente, a primeira etapa do MCPON se caracteriza particularmente pela construção de uma linguagem voltada ao PON, ao passo que define as ferramentas responsáveis para a construção inicial do *front-end* de um compilador para o PON. A primeira etapa do MCPON, porém, se intersecciona com a segunda etapa, uma vez que o início da construção das instâncias do Grafo PON são estabelecidas durante a fase de análises (léxica e sintática).

De maneira geral, conforme apresentado anteriormente na Seção 4.2 e ilustrado na Figura 34, o Grafo PON define um conjunto de entidades instanciáveis que, ao se especializarem e se relacionarem entre si, definem por completo a representação de cada programa com as diretrizes do PON. Por conta disso, o Grafo PON segue um conjunto de diretrizes que deve ser seguido para construir e instanciar grafos especializados padronizados e uniformes, com o objetivo de auxiliar no processo de compilação de programas PON.

Ademais, cada uma das entidades que compõem o Grafo PON apresentam as propriedades necessárias que permitem descrever minuciosamente qualquer programa escrito em PON para uma representação intermediária em forma de um grafo especializado. Na prática, a criação de um grafo especializado que define o estado e o fluxo de execução de um programa, precisa ser composto pontualmente pelas entidades pertinentes mapeadas na análise de um código-fonte e instanciadas de acordo para a correta população do grafo. Nesse âmbito, as próximas subseções detalham as próximas subetapas para a correta construção das instâncias do Grafo PON.

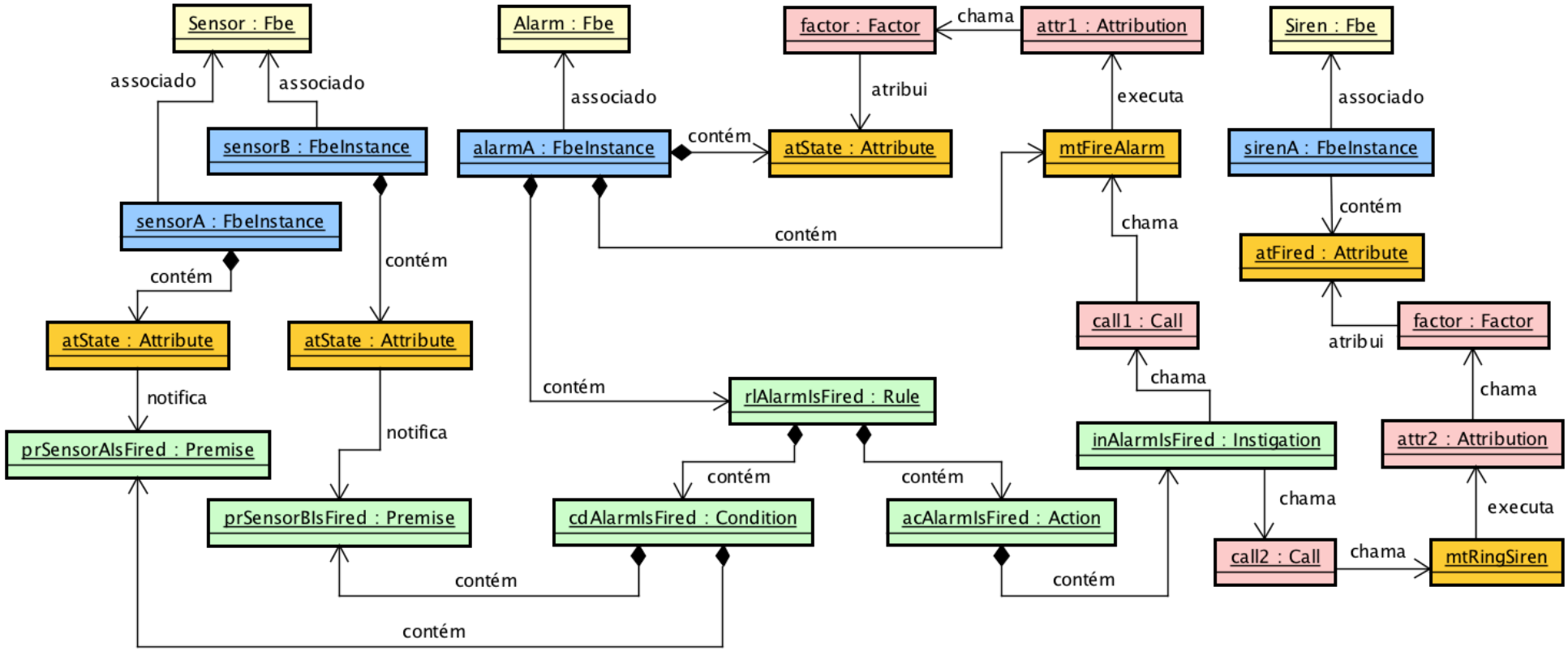
4.3.2.1 Instanciar as entidades e popular instâncias do Grafo PON

Conforme apresentado na subseção anterior, o Grafo PON é constituído por um conjunto de 27 entidades, as quais são supostamente projetadas para serem suficientes para o mapeamento de qualquer programa PON. É importante salientar que as instâncias de elementos específicos (como entidades do tipo *FBE*, ou até mesmo *Rules* internas à uma instância de *FBE*) são criadas em um processo interdependente e posterior ao das análises léxica e sintática. Em tal processo são identificados os elementos que possuem instâncias a serem criadas, as quais são instanciadas e relacionadas com as demais entidades pertinentes.

É importante salientar que as entidades do Grafo PON são *templates* para a construção de grafos especializados e, portanto, precisam ser instanciadas pontualmente de acordo com as características do programa, de modo a representar cada pequena parte deste programa.

Em linhas gerais, ao passo que as entidades vão sendo descobertas e instanciadas, a conexão entre elas também é igualmente estabelecida. Dessa forma, o grafo especializado é constituído com base na interconexão de todas as entidades instanciadas, representando assim um programa integralmente. Nesse âmbito, de modo a ilustrar a construção de uma representação de um programa sob as diretrizes do Grafo PON, o Código 11 apresentado anteriormente em forma de um programa exemplo é reconstruído e ilustrado na Figura 38.

Figura 38: Exemplo de instância do Grafo PON - Programa Alarmes



Fonte: Autoria Própria via Diagrama de objetos em UML

O diagrama de objetos em UML apresentado na Figura 38 ilustra a construção de um grafo especializado de acordo com as diretrizes do Grafo PON a partir de um esboço do programa ‘alarme monitorado’ apresentado no Código 11. Basicamente, as entidades *Alarm, Sensor* e *Siren* são representações de *FBE* (ilustradas pela cor amarela). Cada uma dessas entidades, possuem suas respectivas instâncias neste dado programa (*alarmA*, *sensorA*, *sensorB* e *sirenA*), as quais são representações da classe *FBEInstance* (ilustradas pela cor azul). Na prática, um programa PON pode ter dezenas, centenas ou até milhares de instâncias de *FBE*, as quais seriam todas representadas individualmente e pontualmente no Grafo PON.

Ainda, cada instância de *FBE* pode possuir um conjunto de entidades do tipo *Attribute*, bem como do tipo *Method*, representadas no diagrama pela cor laranja. Ademais, as entidades auxiliares que conectam os elementos notificantes, em especial, as chamadas de métodos (*i.e.*, *Call*) e atribuições (*i.e.*, *Assignment* e *Factor*) são representadas pela cor vermelha. Em suma, conforme apresentado nesta subetapa, as diretrizes para construção do Grafo PON permitem que um programa seja mapeado integralmente na forma de um grafo especializado, produto final este que serve de entrada para a próxima etapa do MCPON.

4.3.2.2 Construir a integração da fase de análise com o Grafo PON

Conforme salientado anteriormente, a primeira e a segunda etapas do método MCPON poderiam ser integradas, de modo que o compilador construa todo o mapeamento de códigos-fonte de uma linguagem PON (função da Etapa 1) para uma instância do grafo especializado (função da Etapa 2). Nesse âmbito, ao passo que o algoritmo definido na Etapa 1 realiza as devidas análises léxica e sintática do código-fonte, o Grafo PON poderia ser construído segundo as diretrizes da Etapa 2.

De modo a apresentar um exemplo sucinto de integração do algoritmo responsável pelas análises com a construção do Grafo PON, o Código 14 apresenta uma implementação de partes do método MCPON desenvolvida com base na linguagem de programação C++ e ferramenta Bison (BISON, 2019). Em tempo, naturalmente, para essa implementação, em especial, o Grafo PON foi implementado em C++ orientado a objetos, respeitando o diagrama de classes dado, formando assim uma espécie de componente utilizável por procedimentos ou funções pertinentes.

Código 14: Exemplo de integração para construção de instâncias do Grafo PON

```
1  fbe               : FBE ID attributes methods {
2                        return graph->createFBE($2, $3, $4);
3                    }
4
5  attributes        : ATTRIBUTES attributes_body END_ATTRIBUTES {
6                        return $2;
7                    }
8
9  attributes_body   : attribute attributes_body
10                   | attribute
11                       attributesTmp->push($1);
12                       return attributesTmp;
13                   }
14
15 attribute         : ID EQUAL VALUE {
16                       return graph->createAttribute($1, $3);
17                   }
```

Fonte: Autoria Própria

Conforme apresentado no Código 14, é possível integrar o algoritmo responsável pelas análises com a criação de instâncias de grafos especializados. Na prática, conforme é sugerido neste trabalho, isso poderia ser desenvolvido com base em um conjunto de métodos que encapsulam a instanciação das entidades do Grafo PON. Em maiores detalhes, a linha 2 está efetivamente criando uma instância de um *FBE*, com base nas informações extraídas das regras sintáticas elaboradas para o exemplo. No exemplo em si, o objeto *graph* representa a materialização do conceito do grafo em linguagem C++, servindo de interface tanto para a instanciação de entidades PON quanto posterior armazenamento das mesmas em uma estrutura de dados. Ademais, é importante reforçar que nada obsta que o desenvolvedor utilize qualquer outra técnica ou linguagem de programação para materializar o método MCPON.

É importante salientar, inclusive, que o Bison (ferramenta utilizada para criação do analisador sintático do exemplo em questão), faz uma análise *bottom-up*. Assim, na prática, esta análise encontra os elementos de um programa de baixo para cima, de forma recursiva, até encontrar o elemento inicial que, no exemplo acima, é o próprio *FBE*. Para isso, inicialmente, o algoritmo vai identificar os *Attributes* e criá-los (linha 16), para posteriormente anexá-los à uma lista (linha 11) e só então retorná-los (linha 6) para a primeira regra sintática (linha 1).

Em suma, esta técnica de compilação permite construir a análise sintática de um programa integrada com a instanciação e população do Grafo PON. Entretanto, algumas questões pontuais não são tratadas a nível de análise léxica e sintática, como

validações de tipo e utilização de entidades não instanciados previamente. Nesse âmbito, uma fase adicional, a de análise semântica é necessária, conforme apresenta a próxima subetapa.

4.3.2.3 Definição das regras semânticas e analisador semântico

Além das análises léxica e sintática na validação de um programa, há ainda a análise semântica com suas peculiaridades. Para essa análise, o desenvolvedor poderia definir um conjunto de validações semânticas de forma a atender as particularidades da linguagem a ser criada. Além disso, o MCPON possibilita a criação de um conjunto de regras de validação semântica, as quais poderiam ser padronizadas e compartilhadas entre os compiladores específicos para o PON. Na prática, tais regras poderiam varrer os grafos especializados de programas PON, de modo a checar se os tipos das operações realizadas entre as entidades são consistentes e compatíveis. Nesse âmbito, o próprio Grafo PON poderia ajudar nesta subetapa, uma vez que ele define um padrão estrutural para todos os grafos de programas PON. Para isso, bastaria apenas criar um conjunto de regras que confrontariam entidades conectadas e validariam a compatibilidade destas.

De modo a ilustrar um possível cenário em que as validações seriam realizadas, o Código 15 apresenta exemplos de validações semânticas em um programa PON.

Código 15: Validações semânticas em um programa PON

```
1  fbe Alarm
2    attributes
3      boolean atOn 10
4      integer atTimer 0
5    end_attributes
6    methods
7      method mtRingTheBell(atTimer = false)
8    end_methods
9  end_fbe
10
11 . . .
12
13 premise prTimerReseted alarm1.atTimer == 0
14 premise prAlarmOn alarm1.atOn == alarm1.atTimer
```

Fonte: Autoria Própria

Conforme apresenta o Código 15, existem três regras de validação semântica definidas para o programa PON fictício, sendo elas:

- *Verificar se o tipo do valor inicial de um Attribute corresponde ao tipo do elemento declarado*. Na linha 3, na qual o *Attribute atOn*, do tipo *boolean*, está sendo inicializado com o valor numérico 10, resultaria em um erro semântico;
- *Verificar se o tipo do valor de uma atribuição corresponde ao tipo do Attribute de destino, em execuções de Methods*. Na linha 7, na qual o *Attribute atTimer*, do tipo *integer*, recebe o valor *false*, do tipo *boolean*, resultaria em outro erro semântico.
- *Verificar se os tipos dos elementos de uma comparação em Premises são do mesmo tipo*. No caso da comparação da linha 13, na qual um *Attribute* do tipo *integer* é comparado com '0', é uma comparação válida semanticamente. Entretanto, a linha 14 apresenta um erro semântico, uma vez que o *Attribute atOn*, do tipo *boolean* não corresponde ao *Attribute atTimer* que é do tipo *integer*.

Conforme é possível observar no trecho de código exemplo, os tipos são importantes tanto na definição de elementos de um programa, como na atribuição de valores para estes, bem como em comparações lógico-causais. A falta de consistência entre eles ocasionaria problemas de execução e, portanto, precisam ser validadas no processo de compilação. Neste âmbito, as regras de validação poderiam percorrer as entidades pertinentes do Grafo PON, de modo a verificar inconsistências, conforme sugerido a seguir:

- *No primeiro erro semântico (linha 3)*: uma regra de validação poderia varrer todos os *FBEIntances* de um programa PON na instância do Grafo PON e verificar se todas as instâncias de *Attribute* possuem um *Type* compatível com o valor inicial deste *Attribute*;
- *No segundo erro semântico (linha 7)*: uma regra de validação poderia varrer todos os *FBEIntances* de um programa PON na instância do Grafo PON e verificar se todas as instâncias de *Method* possuem um *Assignment*, no qual seus dois *Factors* apontem para entidades *Attribute* com *Types* compatíveis;

- *No terceiro erro semântico (linha 14)*: uma regra de validação poderia varrer todos os *FBEIntances* de um programa PON na instância do Grafo PON e verificar se todas as instâncias de *Rules* possuem uma *Condition*, na qual suas *Premises*, bem como as *Premises* de eventuais *Subconditions*, possuem *Expressions* com *Factors* apontando para *Attributes* com *Types* compatíveis.

Conforme é possível observar nos exemplos de regras semânticas, elas possuem um padrão estrutural, uma vez que se baseiam nas diretrizes do Grafo PON, o que viabilizaria a construção de regras semânticas genéricas e padronizadas para qualquer tipo de compilador baseado no MCPON. É importante salientar que a construção fragmentada e desacoplada do Grafo PON é uma característica do próprio PON que, na prática, permite a construção, seguida da análise e ajustes pontuais dos grafos especializados.

Após a conclusão da análise semântica, as entidades do programa estão devidamente mapeadas no grafo especializado criado a partir das diretrizes do Grafo PON. Nesse momento, tal grafo está composto por uma representação completa e fidedigna do código-fonte processado, disponível assim para a geração de código em qualquer linguagem-alvo definida. Entretanto, é possível que existam redundâncias de declarações como no caso de entidades *Premise* com avaliações lógico-causais semanticamente equivalentes, entidades *Instigation* com as mesmas propriedades de execução (*i.e.*, mesma ordem de execução de *Methods*) ou, até mesmo, *Conditions* com o mesmo conjunto de *Premises* em *Rules* distintas. Nesse âmbito, a solução para isso poderia ser resolvida com regras de otimização específicas, com o objetivo de eliminar redundâncias desta natureza, função essa resolvida na Etapa 3 do MCPON.

4.3.3 Construção de Otimizadores

Em linhas gerais, a Etapa 3 do MCPON possui como objetivo principal nortear o processo de construção de otimizadores. Para isso, o desenvolvedor poderia construir regras de otimização de modo a reduzir ou quiçá eliminar as entidades redundantes presentes na estrutura dos grafos gerados nas etapas anteriores, bem como preparar o grafo para a fase de geração de código, particularmente por meio de otimizações específicas. Nesse sentido, a Etapa 3 estabelece duas categorias de

otimizadores. De um lado, do *front-end*, constitui-se os otimizadores genéricos, independentes de *target*. Do outro lado, do *back-end*, por sua vez, constitui-se os otimizadores especializados, dependentes do *target*. Nesse âmbito, as próximas subseções apresentam algumas diretrizes para a construção dos otimizadores desta etapa.

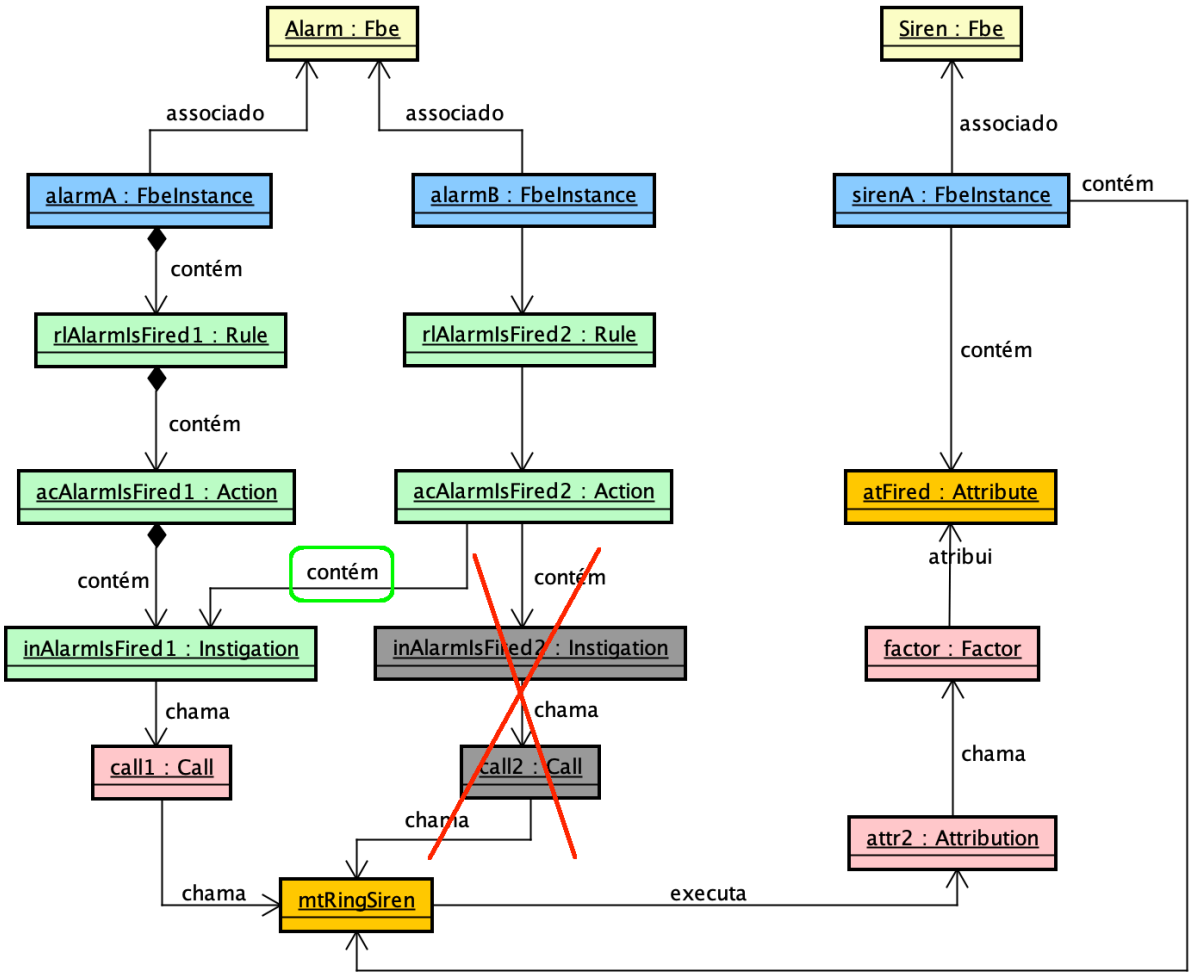
4.3.3.1 Criação de otimizadores genéricos independentes de *target*

Em linhas gerais, apesar de o PON ser teoricamente não-redundante, a estrutura de um programa pode levar a duplicação de entidades do modelo, principalmente quando as entidades são identificadas de forma implícita, ou seja, quando estas não estão explicitamente escritas no programa enquanto entidades, mas sim enquanto código. Isso ocorre porque a população dos grafos acontece de acordo com as respectivas ocorrências das entidades durante a análise de cada programa, as quais podem ser repetidas ao longo de um programa.

Como um exemplo da redundância citada, considera-se uma determinada expressão lógico-causal (*e.g., a == 2*), repetida ao longo do código-fonte de um programa PON. Para o PON, teoricamente, tais expressões seriam transformadas em uma única conexão entre o *Attribute* e a *Premise* em questão, compartilhando o estado dessa *Premise* com todas as *Conditions* pertinentes. Entretanto, a organização de um grafo durante as primeiras duas etapas do método não validam essa redundância, duplicando a entrada de tal *Premise* no grafo, portanto.

Neste âmbito, de modo a ilustrar possíveis redundâncias otimizáveis, a Figura 39 apresenta um exemplo de grafo especializado com um exemplo de redundância pontual.

Figura 39: Exemplo de redundâncias em um grafo especializado



Fonte: Autoria Própria via Diagrama de objetos em UML

A Figura 39 ilustra, na forma de um diagrama de objetos, a essência de parte de um programa PON em forma de grafo especializado, o qual apresenta um problema de redundância. Conforme apresenta o diagrama, tanto a *Instigation inAlarmIsFired1* quanto a *Instigation inAlarmIsFired2* (e suas entidades auxiliares *Call*) fazem chamadas ao *Method mtRingSiren*, sem nenhuma diferença em suas propriedades internas.

Neste âmbito, uma regra de otimização poderia eliminar a existência de uma destas entidades e reconectar a entidade *acAlarmIsFired2* com a *Instigation inAlarimIsFired1*, mantendo a essência do programa intacta e mais enxuta. Na prática, tal problema dificilmente seria identificado na análise sintática de um programa, o qual instanciaria as entidades de acordo com suas respectivas ocorrências na análise deste programa. Sendo assim, a solução para isso poderia ser resolvida com regras

de otimização específicas, com o objetivo de eliminar redundâncias desta natureza nos grafos especializados baseados no Grafo PON.

Em suma, um conjunto de regras de otimização poderia ser criado de maneira genérica, ou seja, independente de *target*, com o objetivo de otimizar diversas linguagens e compiladores baseados no MCPON. Na prática, a construção de tal conjunto de regras de otimização pode ser viabilizada por meio de um algoritmo independente cuja entrada seria um grafo especializado, construído nas etapas anteriores e, como saída, um grafo otimizado.

É importante observar que esta subetapa finaliza a construção do *front-end* de um compilador especializado em PON. Com base nesse *front-end*, qualquer programa PON de entrada seria devidamente analisado, mapeado e otimizado em uma estrutura de dados única e uniforme, sob as diretrizes do Grafo PON. Todavia, em alguns casos, como em ambientes paralelos e distribuídos ou, até mesmo, em ambientes de hardware digital, os quais poderiam requerer de configurações adicionais e específicas para a sua correta execução, ainda demandariam de uma subetapa de otimização adicional. Nesse âmbito, tal subetapa é apresentada na próxima subseção.

4.3.3.2 Criação de otimizadores especializados dependentes de *target*

Com base no grafo otimizado da subetapa anterior seria possível gerar o código-alvo para qualquer *target* em qualquer linguagem alvo. Entretanto, existem casos em que é necessário realizar configurações adicionais e específicas prévias a geração de código. Nesse âmbito, o MCPON prevê uma subetapa adicional, responsável pela criação de otimizadores especializados dependentes de *target*. Tal subetapa é opcional e pode ser dispensada. Em caso de necessidade desta subetapa, a construção do *back-end* especializado para determinado *target* teria início a partir destas regras de otimização.

Essa subetapa, porém, não tem impacto na fase de análise do compilador e, por consequência, não afeta a escrita do código-fonte de programas PON. Isso conforma com uma das características das linguagens baseadas no MCPON que prevê que estas sejam genéricas e compatíveis com qualquer *target* de geração e código. Nesse âmbito, os programas PON desenvolvidos deveriam compilar para qualquer *target* e não estariam atrelados a um *target* específico.

Para isso, inicialmente é necessário definir a plataforma-alvo para qual se destina o processo de compilação, bem como a linguagem-alvo para a qual os grafos otimizados serão traduzidos na próxima etapa de geração de código. Além disso, é importante definir as particularidades do *target* a ser desenvolvido no tocante à possibilidade de execução paralela ou, até mesmo, distribuída. Neste âmbito, o desenvolvedor deveria se atentar as particularidades da plataforma-alvo para verificar as possibilidades de implementação concorrente, paralela ou distribuída, particularidades estas, importantes para a essência do PON.

Isto posto, recomenda-se que para esta subetapa seja definido um arquivo de configuração, o qual pode ser interpretado ao final da fase de análise realizada pelo *front-end*, associando as definições expressas no arquivo de configuração às regras de otimização especializadas. A título de exemplo, um arquivo de configuração em um ambiente paralelo e concorrente, poderia definir quantas *threads* e núcleos de processamento seriam responsáveis pela execução de determinado programa. Ainda, tal arquivo poderia ser ainda mais minucioso, ao definir em qual *thread* ou núcleo de processamento, cada conjunto de entidades PON deveria executar, possibilitando um processo de micro-otimizações pontuais para o *target* específico.

Em um ambiente distribuído, por sua vez, poderia se utilizar da mesma técnica para definir, por exemplo, o protocolo de comunicação e o endereço de cada um dos nós de processamento disponíveis, bem como quais grupos de entidades seriam processados em cada nó. Por fim, em um ambiente de hardware digital poderiam ser definidas questões como a associação de endereços de memória específicos para cada entidade instanciada, assim como definições outras, como número de unidades lógicas disponíveis que poderiam até ser utilizadas como medida para reduzir o escopo de um programa ou impedir a sua compilação.

De qualquer forma, essa subetapa se mostra pertinente, principalmente em ambientes de natureza mais complexa e com possibilidades de microajustes. Nesse âmbito, ao fim desta subetapa estariam disponíveis um ou mais grafos otimizados, dependendo do caso, e as devidas configurações definidas para a próxima subetapa, responsável pela geração efetiva de código. Nesse âmbito, a próxima subetapa do MCPON, tem como entrada uma estrutura otimizada e preparada para a correta geração de código para o *target* escolhido.

4.3.4 Geração de código

Em linhas gerais, a quarta etapa do MCPON consiste na construção efetiva de geradores de código, alinhado com a etapa anterior, na qual foram gerados grafos otimizados para esse fim. Basicamente, tais grafos servem de entrada para o processo de tradução e geração de código. Neste âmbito, as próximas subseções apresentam, em detalhes, o processo de iteração dos grafos para a posterior geração de código-alvo.

4.3.4.1 Iterar instâncias do Grafo PON

De maneira geral, a quarta etapa do MCPON contempla o processo de geração de código. É importante observar que a etapa de geração de código não precisa necessariamente estar associada a tradução efetiva do código PON para um código de máquina específico da plataforma-alvo em questão. Isto é, a etapa de tradução poderia ser voltada especificamente para uma linguagem de programação especializada na plataforma-alvo. Neste âmbito, a tradução dos grafos otimizados do MCPON seria voltada particularmente para esta linguagem específica, para uma posterior compilação dos códigos gerados para o código de máquina[10]. Não obstante, o fluxo de execução dessa tradução deveria respeitar os princípios do PON no tocante à orientação a entidades notificantes, de modo a preservar a essência do paradigma.

Ainda, em tempo, o desenvolvedor poderia construir seu compilador de maneira a traduzir diretamente os grafos otimizados para código de máquina da plataforma-alvo. Em alguns casos, também é possível traduzir para representações intermediárias, como no caso da linguagem Java que é baseada em máquinas virtuais e em uma linguagem intermediária (*i.e.*, *bytecode*). Ademais, algumas plataformas poderiam basear sua execução em *scripts*, as quais geralmente se apresentam de maneira interpretada. Em suma, a definição de uma plataforma-alvo, bem como da

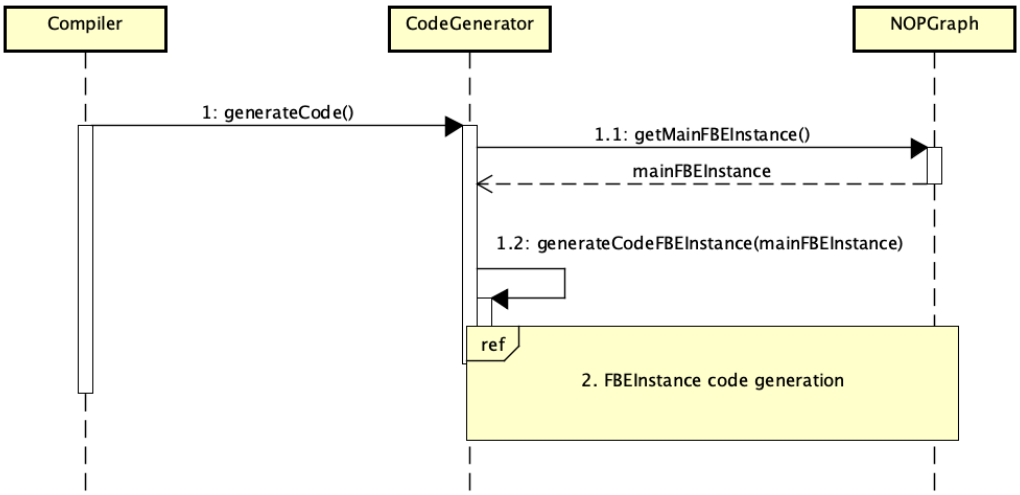
[10] É importante ressaltar que o método MCPON prevê apenas a tradução do Grafo PON para códigos-alvo específicos. Uma vez que tais códigos se baseiem em linguagens de alto-nível, a compilação posterior para código de máquina fica a cargo de compiladores específicos. Neste âmbito, não caberia ao processo de compilação do MCPON abranger a compilação completa do processo, como nesses casos em que ocorre a tradução para outras linguagens. Entretanto, o desenvolvedor poderia integrar em seu compilador as bibliotecas ou ferramentas necessárias para este fim.

linguagem-alvo para qual a qual se destina o compilador não possui restrições, com uma ressalva de que as características do PON sejam preservadas tanto quanto possível.

De maneira geral, para transformar grafos especializados em códigos-alvo destinados à *targets* específicos, o desenvolvedor deveria construir um algoritmo que iteraria sobre todas as instâncias de entidades do Grafo PON. Conforme o algoritmo navegaria nos grafos, para cada entidade encontrada ao longo do processo, este deveria gerar um código especializado para reconstruir as características destas entidades no *target* desejado.

A subetapa de navegar nos grafos especializados também deveria seguir um padrão, uma vez que a estrutura do Grafo PON permite a navegação de forma recursiva e em profundidade. Neste âmbito, os diagramas de sequência a seguir apresentam um processo de navegação que possibilitaria a reconstrução dos grafos em qualquer plataforma-alvo.

Figura 40: Processo de iteração do Grafo PON

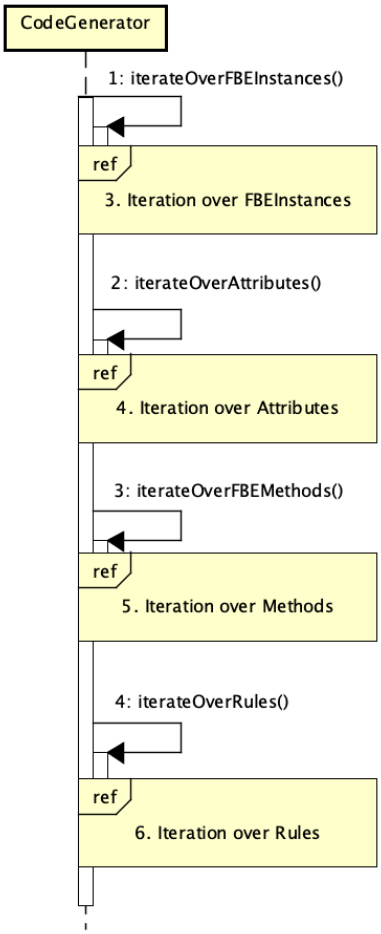


Fonte: Autoria Própria via Diagrama de sequência em UML

O diagrama de sequência ilustrado na Figura 40 apresenta o processo inicial de iteração do Grafo PON. Para isso, primeiramente, a objeto responsável pelo compilador específico (*e.g.*, instância da classe *Compiler*) poderia chamar um método *generateCode* para iniciar o processo como um todo. De maneira geral, um grafo especializado para um determinado programa PON, possui uma instância principal de um *FBEInstance* (*i.e.*, *mainFBEInstance*). Com base nesta, a classe responsável pela geração de código poderia reconstruir tal entidade no código-alvo da plataforma a qual

se destina o compilador. Cada entidade do tipo *FBEInstance* possui um conjunto de relações com as entidades *Attributes*, *Methods* e *Rules*. Neste âmbito, tais relações deveriam ser reconstruídas no *target*-alvo. Este processo é apresentado no diagrama de sequência ilustrado na Figura 41.

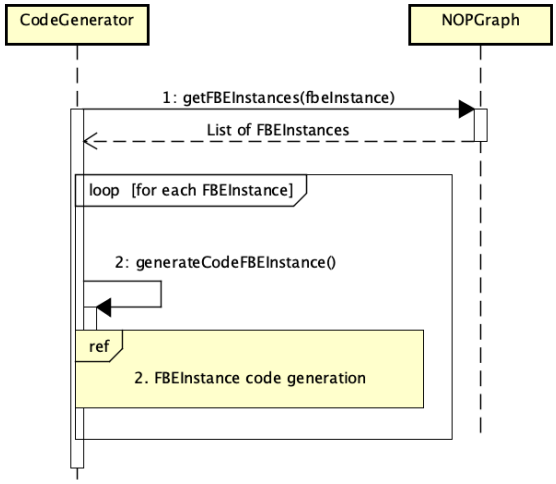
Figura 41: Geração de código para a entidade *FBEInstance*



Fonte: Autoria Própria via Diagrama de sequência em UML

A Figura 41 ilustra o diagrama de sequência que mapeia a reconstrução ordenada das entidades relacionadas à entidade *FBEInstance*. Na prática, o Grafo PON é basicamente formado por composições de instâncias de *FBE*, as quais podem possuir N níveis de relacionamento com outras instâncias de *FBE*. Além disso, uma *FBEInstance* pode possuir um conjunto de *Attributes*, um conjunto de *Methods* e um conjunto de *Rules*. Neste âmbito, basicamente o algoritmo de geração de código deveria iterar sobre estes conjuntos de entidades para reconstruir a estrutura de uma instância de *FBE*. Para isso, os diagramas de sequência ilustrados nas Figuras 45, 47 e 48 contemplam o processo de iteração de tais conjuntos.

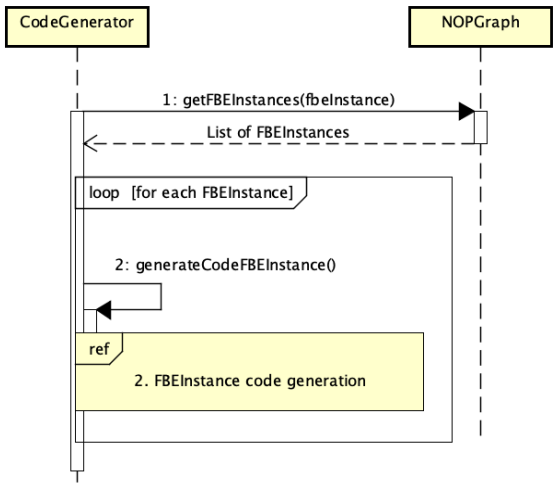
Figura 42: Processo de iteração sobre instâncias de *FBE*



Fonte: Autoria Própria via Diagrama de sequência em UML

A Figura 42 ilustra o processo de iteração de instâncias de *FBE*. Conforme dito anteriormente, as instâncias de *FBE* podem possuir N níveis de composição. Neste âmbito, o algoritmo poderia ser construído de forma recursiva, de modo a recuperar e reconstruir cada camada de instâncias de *FBE* a cada iteração. Na prática, tais instâncias deveriam ser recuperadas do Grafo PON e, para cada entidade vinculada, gerar a reconstrução pertinente. Além das chamadas recursivas entre instâncias de *FBE*, cada qual deveria construir suas particularidades, as quais são mapeadas nos diagramas a seguir.

Figura 43: Processo de iteração sobre um conjunto de *Attributes*

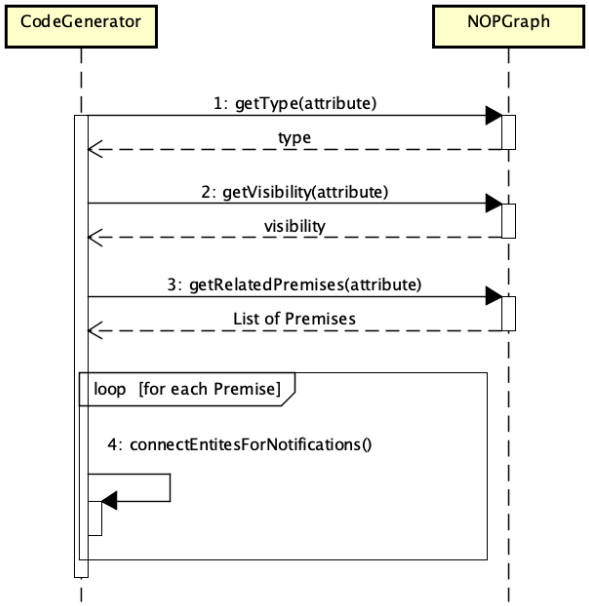


Fonte: Autoria Própria via Diagrama de sequência em UML

A Figura 43 ilustra o diagrama de sequência do processo de iteração sobre o conjunto de *Atributes* de uma dada instância de *FBE*. Na prática, este conjunto de

Attributes deveria ser recuperado do Grafo PON e, para cada *Attribute*, gerar a reconstrução pertinente deste de acordo com as particularidades desta entidade. Neste âmbito, a Figura 44 apresenta as principais características de um *Attribute*.

Figura 44: Processo de criação de *Attributes*

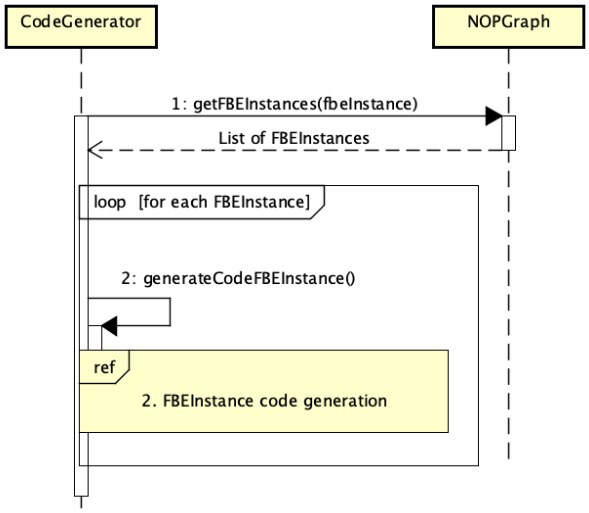


Fonte: Autoria Própria via Diagrama de sequência em UML

Conforme ilustra o diagrama de sequência apresentado na Figura 44, a reconstrução da entidade *Attribute* está intimamente ligada ao Grafo PON, uma vez que as características desta entidade estão todas mapeadas no grafo. Para isso, inicialmente, o gerador de código deveria recuperar as propriedades de cada entidade como o *Type* e a *Visibility* deste *Attribute* e construir o código-alvo de acordo com tais propriedades. Além disso, as conexões de *Attributes* com *Premises* também são mapeadas no grafo e podem ser resgatadas neste passo, possibilitando a construção destas representações na plataforma-alvo a qual se destina o compilador.

Ademais, no âmbito de iterações sobre as entidades principais de uma instância de *FBE*, está o mapeamento dos *Methods*, conforme apresenta o diagrama ilustrado na Figura 45.

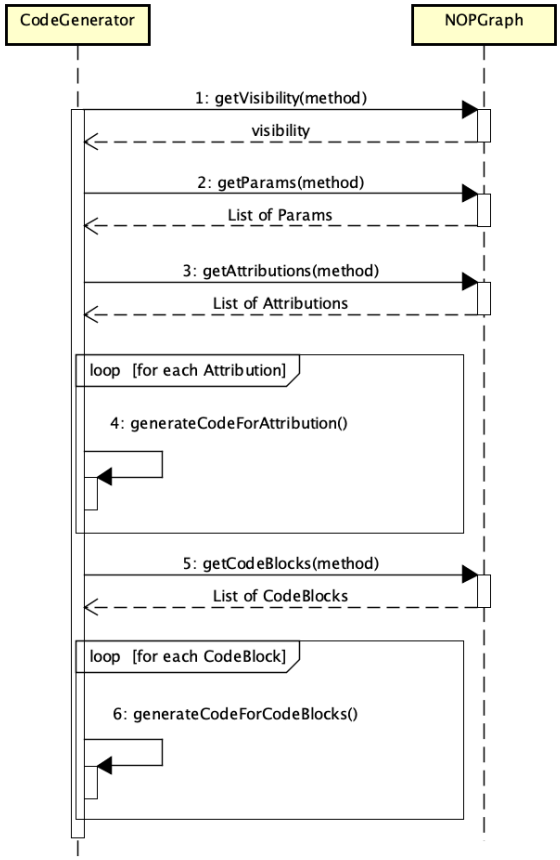
Figura 45: Processo de iteração sobre um conjunto de *Methods*



Fonte: Autoria Própria via Diagrama de sequência em UML

Assim como a iteração para reconstrução do conjunto de *Attributes*, o Grafo PON mapeia o conjunto de *Methods* associados a cada instância de *FBE*. Neste âmbito, o processo de iteração de *Methods* segue a mesma lógica, conforme ilustra o diagrama de sequência apresentado na Figura 46.

Figura 46: Processo de criação de *Methods*

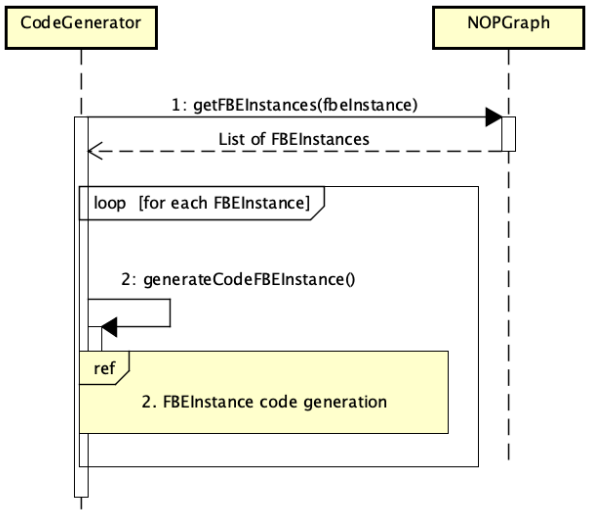


Fonte: Autoria Própria via Diagrama de sequência em UML

Conforme apresenta o diagrama de sequência ilustrado na Figura 46, a composição de um *Method* basicamente é representada pelas entidades *Visibility*, assim como por suas características execucionais, sejam elas mapeadas por meio de um conjunto de *Assignments*, como por um conjunto de *CodeBlocks*. Tais características também podem se apoiar em um conjunto opcional de *Params*. Na prática, em geral, um *Method* é uma entidade flexível que pode assumir diferentes características ao se associar à diferentes entidades. Neste âmbito, o desenvolvedor deveria construir um gerador de código capaz de identificar e tratar cada possibilidade de combinação de *Methods* distintos de modo a construir o código-alvo adequadamente.

Dando sequência ao processo de construção de uma instância de *FBE,* finalmente, são construídas as *Rules* que mapeiam o processo lógico-causal de um programa. Neste âmbito, os diagramas de sequência a seguir ilustram o processo.

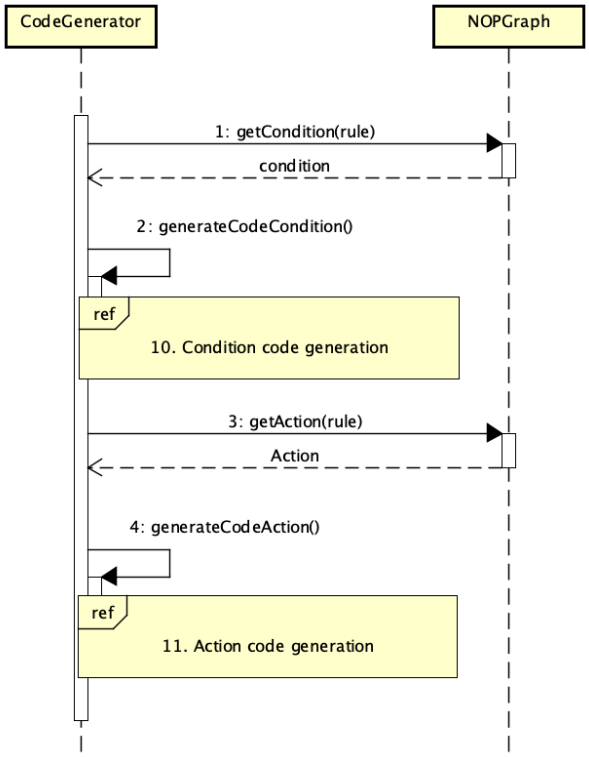
Figura 47: Processo de iteração sobre um conjunto de *Rules*



Fonte: Autoria Própria via Diagrama de sequência em UML

Igualmente às iterações de *Attributes* e *Methods*, o Grafo PON mapeia o conjunto de *Rules* associados a cada instância de *FBE*. Neste âmbito, o processo de iteração de *Rules* segue a mesma lógica, conforme ilustra o diagrama de sequência apresentado na Figura 48.

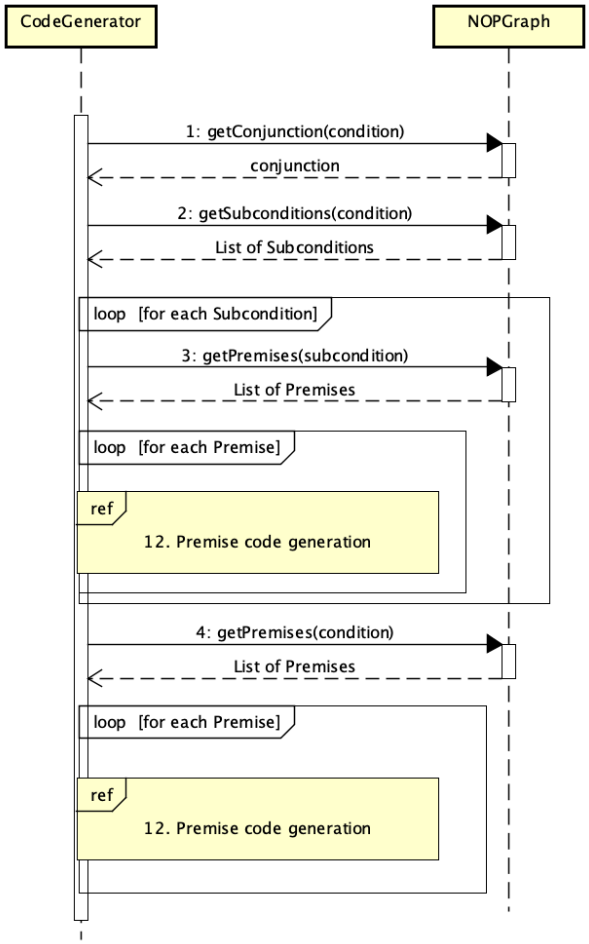
Figura 48: Processo de criação de *Rules*



Fonte: Autoria Própria via Diagrama de sequência em UML

O diagrama de sequência ilustrado na Figura 48 apresenta o fluxo de iteração de uma *Rule*. Basicamente recupera-se do Grafo PON tanto a entidade *Condition* quanto a entidade *Action*, associadas à esta *Rule*. Com base em cada qual, delega-se a geração de código destas entidades para processos pontuais. Tais processos são ilustrados nos diagramas das figuras 51 e 53.

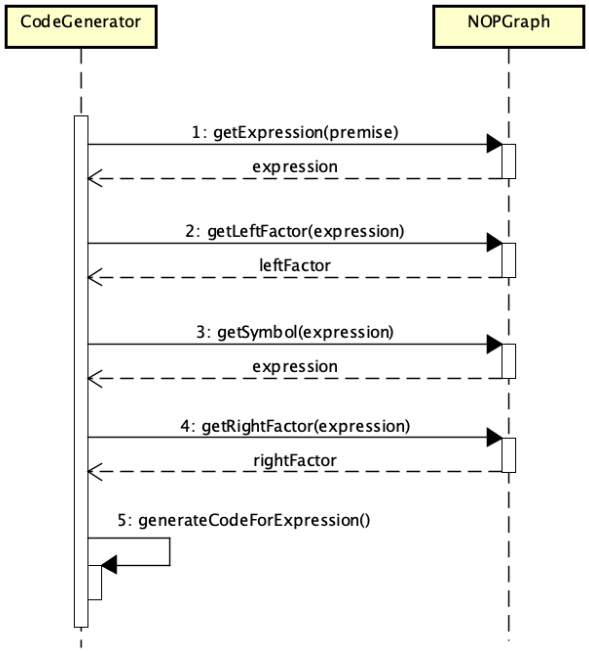
Figura 49: Processo de criação de *Conditions*



Fonte: Autoria Própria via Diagrama de sequência em UML

O diagrama de sequência ilustrado na Figura 49 apresenta o processo de criação de *Conditions*. Inicialmente, recupera-se do Grafo PON a entidade *Conjunction*, a qual basicamente define o tipo de conexão lógica entre as demais entidades. Na sequência, tal *Condition* poderia acessar um conjunto de *Subconditions*, e por meio destas, um conjunto de *Premises*. Ademais, em uma *Condition*, também é possível acessar um conjunto de *Premises* diretamente. Para ambos os casos, o algoritmo deveria delegar a geração de código de *Premises* para um processo específico. Tal processo é ilustrado no diagrama da Figura 50.

Figura 50: Processo de criação de *Premises*

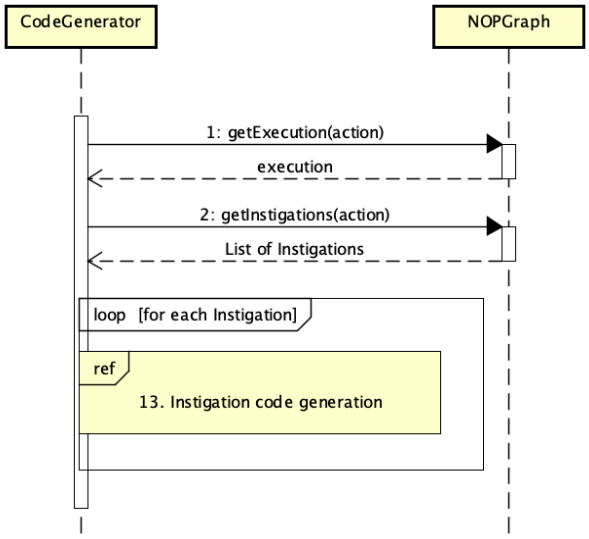


Fonte: Autoria Própria via Diagrama de sequência em UML

O diagrama de sequência ilustrado na Figura 50 apresenta a criação de *Premises*. Cada entidade possui um vínculo com a entidade *Expression*, a qual precisa ser recuperada do Grafo PON. Com base nesta, é possível extrair os demais elementos pertinentes para a construção de tal expressão, tais como o fator do lado esquerdo, o símbolo de comparação e o fator do lado direito da expressão. Por fim, com acesso à todas as características, o algoritmo é capaz de construir pontualmente esta expressão.

Dando sequência ao processo de construção de uma *Rule,* a Figura 51 ilustra o diagrama de sequência do processo de criação de *Actions*.

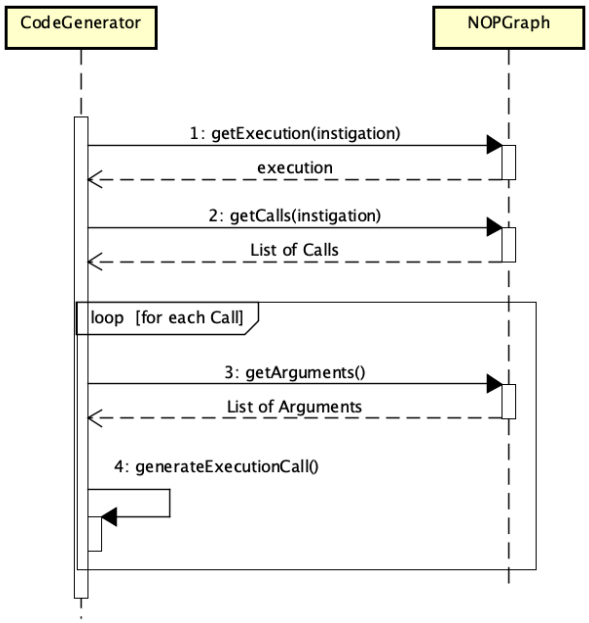
Figura 51: Processo de criação de *Actions*



Fonte: Autoria Própria via Diagrama de sequência em UML

Conforme apresenta o diagrama de sequência ilustrado na Figura 51, inicialmente recupera-se a entidade *Execution* do Grafo PON, de maneira a definir o modo de execução e o encadeamento das *Instigations* associadas. Tais *Instigations* são igualmente recuperadas do Grafo PON para serem construídas em um processo independe, o qual é apresentado no diagrama ilustrado na Figura 52.

Figura 52: Processo de criação de *Instigations*



Fonte: Autoria Própria via Diagrama de sequência em UML

Conforme apresenta o diagrama de sequência ilustrado na Figura 52, tal qual a *Action*, a *Instigation* também faz uso da entidade *Execution* para definir o mecanismo

de execução das chamadas de *Methods*. Neste âmbito, recupera-se do Grafo PON um conjunto de *Calls*, de maneira a definir e organizar as chamadas de *Method*, com base em um conjunto de *Arguments*, os quais também são recuperados do Grafo PON.

Em suma, esta coleção de diagramas de sequência apresentam as principais características para a correta exploração do Grafo PON. Na prática, a 'raiz' do grafo, por assim dizer, é a instância principal do *FBE Main*. Neste caso, todas as demais entidades estão hierarquicamente abaixo desta entidade principal e podem ser encontradas por meio da navegação ordenada apresentada. Com base nisso, o desenvolvedor poderia construir o código-alvo de qualquer *target*, conforme exemplifica a próxima subseção.

4.3.4.2 Construção de geradores de código

Conforme apresentado na subseção anterior, as instâncias do Grafo PON poderiam ser exploradas completamente por meio das diretrizes estabelecidas, como base na busca em profundidade e iteração recursiva. Na prática, a construção de geradores de código deveria levar em consideração tais diretrizes e associar ao passo de iteração um conjunto de funções que criariam os códigos-alvo efetivamente.

Para elaborar tal ferramenta, o desenvolvedor poderia construir um algoritmo de iteração especializado, com base nas diretrizes do Grafo PON e com base em métodos tradicionais de escrita de arquivo. Na prática, com o auxílio de bibliotecas de entrada e saída de dados, bastaria simplesmente construir o código adequadamente e iterativamente. Neste sentido, o Código 16 apresenta um algoritmo que exemplifica a iteração com posterior escrita em arquivo.

Código 16: Exemplo de algoritmo para iteração do Grafo PON

```
1  void CodeGenerationExample::iterateOverRules(Instance *instance, int level) {
2
3    map<string, Rule*> *rules = NOPGraph::getRules(instance);
4
5    for (map<string, Rule*>::iterator it=rules->begin(); it!=rules->end(); ++it) {
6
7      Rule *rule = it->second;
8
9      Condition *condition = NOPGraph::getConditions(rule);
10
11     map<string,Subcondition*>*subconditions=NOPGraph::getSubconditions(condition);
12
13     for (map<string, Subcondition*>::iterator it = subconditions->begin();
14      it != subconditions->end(); ++it) {
15
16       Subcondition *subcondition = it->second;
```

```
17
18        map<string, Premise*> *premises = NOPGraph::getPremises(subcondition);
19
20        for (map<string, Premise*>::iterator it = premises->begin();
21         it != premises->end(); ++it) {
22
23          Premise *premise = it->second;
24          generateCodePremise(premise, level);
25
26        }
27      }
28
29      map<string, Premise*> *premises = NOPGraph::getPremises(condition);
30
31      for (map<string, Premise*>::iterator it = premises->begin();
32       it != premises->end(); ++it) {
33
34        Premise *premise = it->second;
35        generateCodePremise(premise, level);
36
37      }
38    }
39  }
```

Fonte: Autoria Própria

Em linhas gerais, o Código 16 apresenta um exemplo de iteração do Grafo PON em linguagem C++, mais especificamente sobre o conjunto de *Rules* de uma instância de *FBE*. Conforme é possível observar no trecho de código, inicialmente, o conjunto de *Rules* é recuperado do Grafo PON (linha 3). Com base neste, a iteração basicamente percorre tal conjunto de *Rules* (linha 5) recuperando as entidades relacionadas e direcionando as chamadas de métodos para funções específicas de geração de código (linhas 24 e 35). Tais funções são apresentadas no Código 17, em um trecho de código exemplo, específico para a construção da entidade *Premise*.

Código 17: Exemplo de geração de código para entidade *Premise*

```
1  void CodeGenerationExample::generateCodePremise(Premise *premise, int level) {
2    level++;
3    sstream << getLevel(level) << "Premise* " << premise->getName() << "(";
4    Expression *expression = NOPGraph:: getExpression(premise);
5    Factor *leftFactor = expression->getLeftFactor();
6    if (leftFactor->getFactorId() == Factor::ELEMENT_FACTOR) {
7      ElementFactor *element = (ElementFactor*)leftFactor;
8      sstream << element->getInstance()->getName();
9      sstream << "->";
10     sstream << element->getAttribute()->getName();
11   } else {
12     sstream << leftFactor->getStringValue();
13   }
14
15   . . .
16 }
```

Fonte: Autoria Própria

Conforme apresenta o Código 17, a escrita de um arquivo de código-alvo basicamente faz uso da biblioteca *sstream* conforme apresenta a linha 5 do trecho de código. Cada detalhe de uma *Premise*, ao longo do percorrimento de suas características, poderia ser traduzido diretamente para o fluxo de saída, mapeado em

um arquivo de código-alvo pertinente. Ademais, conforme apresenta o trecho de código em questão, o desenvolvedor possui total liberdade para criar técnicas de compilação próprias que auxiliem no processo de compilação como um todo, tais como facilitadores de indentação, variáveis temporárias para mapear valores ou, até mesmo, estruturas de dados e funções auxiliares.

Em suma, como resultado final desta etapa, espera-se a tradução efetiva e completa de um grafo especializado em um conjunto de arquivos de código compilado para a plataforma-alvo definida igualmente nesta etapa. Com base em tais arquivos, o desenvolvedor poderia fazer a conexão destes com a ferramenta de compilação da plataforma-alvo, a qual se destina tal materialização. Não obstante, o *target* poderia ser traduzido diretamente para código de máquina, *scripts* ou, até mesmo, representações intermediárias outras, de acordo com as características definidas especialmente para esta etapa do método MCPON.

4.3.5 Validação do processo

Após a etapa de geração de código, o processo de compilação já está completo. Entretanto, uma etapa fundamental é a de verificação e validação do processo como um todo. De maneira a manter a consistência e qualidade dos compiladores gerados, o método MCPON apresenta uma etapa adicional, especificamente para a validação do processo.

Outrossim, é importante reforçar que os compiladores representam uma das mais importantes infraestruturas para construção de software. Erros nesta etapa, principalmente quando se tratarem de “erros silenciosos” ou, até mesmo, geração de código inconsistente, são problemáticos e inadmissíveis. O fato se agrava quando um programa compilado se comporta de forma incorreta mesmo quando o código-alvo parecer inteiramente correto. Apesar do senso comum dizer que “nunca é problema do compilador”, bugs em compiladores são relativamente comuns. Neste âmbito, um roteiro de testes é uma forma de garantir a qualidade dos compiladores.

De maneira geral, o processo de testes em compiladores tradicionais normalmente é implementado por algoritmos interdependentes, bem como técnicas de teste de software tradicionais. Todavia, no método MCPON é possível automatizar os testes com base em um conjunto de pequenos programas em forma de instâncias

de grafos especializados, os quais apresentam de forma distinta as características do PON e de suas respectivas propriedades intrínsecas.

Com base nesta coleção de casos de teste é possível mensurar a integralidade e abrangência dos conceitos do PON nos *targets* desenvolvidos. Esta subetapa é apresentada na Seção 4.3.5.1. Por fim, outro meio de validar os *targets* desenvolvidos é a compilação de programas completos para o PON. Em suma, os programas completos seriam uma espécie de teste mais abrangente, uma vez que contemplariam várias características da linguagem com objetivos claros para as entradas e saídas de tais programas. Isto, de fato, viabiliza a validação completa de execução do programa compilado como um todo, de maneira a verificar se o mesmo está executando apropriadamente de acordo com os princípios do paradigma. Mais detalhes sobre esta última subetapa estão na Seção 4.3.5.2.

4.3.5.1 Testes de integridade

De maneira geral, conforme mencionado, a validação do processo de compilação do MCPON poderia se amparar em um conjunto de casos de teste para a verificação da abrangência dos conceitos implementados nos *targets* em questão. Este conjunto de casos de teste poderia ser desenvolvido de maneira genérica, possibilitando sua utilização na validação de qualquer *target* baseado no MCPON. Isto por si só justifica a implementação de uma ferramenta que poderia ser comum a todos os compiladores baseados no método MCPON.

Nesta ferramenta seria possível desenvolver algum tipo de mecanismo para a atribuição de valores diretamente nos *Attributes* dos programas compilados. Isto serviria basicamente para validar as entradas e saídas da execução do mecanismo de inferência do PON no *target* em questão, de modo a verificar a consistência das notificações e, até mesmo, o determinismo de execução dos pequenos programas presentes na coleção de casos de teste.

Neste sentido, sugere-se ao desenvolvedor criar um programa ‘testador’ capaz de ser integrado ao *target* desenvolvido. Tal integração poderia ser realizada diretamente via software, com base no padrão de projeto *Facade* ou, até mesmo, por meio de uma *API*. Tal integração deveria disponibilizar meios de permitir ao programa ‘testador’ introduzir notificações pontuais nos programas gerados. Na prática, tal interface deveria possibilitar tanto alterar o valor dos *Attributes* de um programa,

quanto ler o valor atual dos mesmos, de modo a validar toda a cadeia de notificações. Essa validação se daria basicamente verificando se a alteração de valores de entrada de um dado programa levará ao estado final desejado após a aplicação dos testes.

4.3.5.2 Compilação de programas completos

Além do conjunto de casos de teste composto por pequenos programas que contemplam as características pontuais do PON, seria possível fazer uma validação mais abrangente do *target* desenvolvido, particularmente por meio de programas completos. Neste âmbito, o desenvolvedor poderia utilizar o mesmo mecanismo de adaptação utilizado na subetapa anterior, visando a integração com os programas desenvolvidos, de maneira a permitir que atribuições possam ser geradas pelo programa 'testador'. Desta forma, tal programa teria acesso aos estados internos dos *Attributes* de modo a verificar as alterações pontuais nos estados de cada qual, com o objetivo de validar a consistência da máquina de inferência especializada no programa testado.

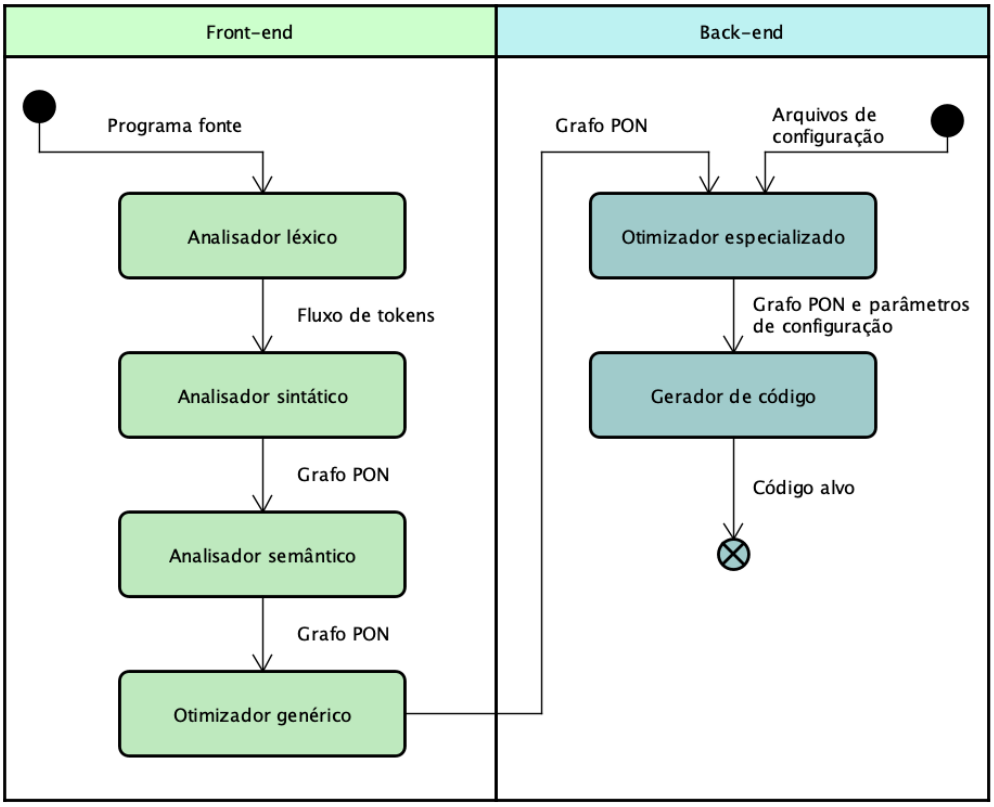
Basicamente este conjunto de testes completos poderia validar a integração de múltiplas propriedades do PON em conjunto. Para isso, tais programas deveriam estar em forma de grafos otimizados, com todas as características do programa mapeados nestes. Além disso, outra possibilidade no tocante à testes de programas completos, seria a validação das propriedades elementares do PON, como testes de desempenho. Tais testes poderiam ser implementados com base em geradores de programas, os quais basicamente poderiam criar grafos completos, com o objetivo de estressar as particularidades do PON, como um grande número de alterações de *Attributes*, com certa redundância ou não. Com base nisso seria possível verificar o impacto das notificações na plataforma-alvo escolhida.

4.4 ESTRUTURA DE COMPILADORES BASEADOS NO MCPON

De maneira geral, a estrutura de compiladores baseados no MCPON é formada tanto pelas camadas de *front-end* quanto de *back-end*. Em geral, as etapas do processo de compilação são fortemente baseadas no Grafo PON, o qual basicamente serve como representação intermediária uniforme para a maioria das

etapas de um compilador para o PON. Nesse sentido, a Figura 53 ilustra a estrutura geral de compiladores baseados no método MCPON.

Figura 53: Estrutura de compiladores baseados no MCPON



Fonte: Autoria Própria

Conforme ilustra a Figura 53, o *front-end* de um compilador baseado no método MCPON consiste essencialmente nas etapas de análise (*i.e.,* análise léxica, análise sintática e análise semântica) e na etapa de otimização independente de *target*. Inicialmente, o compilador recebe como entrada um programa fonte, o qual passa pelo analisador léxico para a extração do fluxo de *tokens*. Posteriormente, o analisador sintático cria a estrutura geral de uma instância do Grafo PON. Na sequência, o analisador semântico faz as devidas checagens deste grafo especializado. Por fim, o otimizador genérico faz todos os ajustes e otimizações independentes de *target* necessárias em tal grafo, finalizando o processo de compilação da camada de *front-end* do compilador.

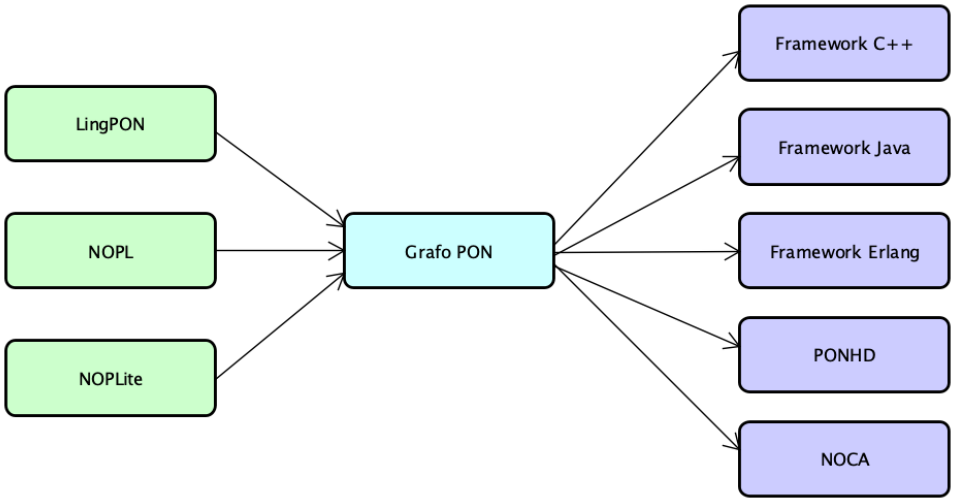
A seu turno, o *back-end* de um compilador baseado no método MCPON consiste basicamente no otimizador especializado dependente do *target* e na etapa de geração de código. Inicialmente, o otimizador especializado recebe tanto o grafo otimizado do *front-end* quanto um arquivo de configuração específico do *target*

desejado. Por fim, o gerador de código faz a geração de código alvo com base no grafo otimizado e nos parâmetros de configuração definidos, os quais podem levar em conta tanto características da linguagem-alvo quanto da própria arquitetura visada.

Como uma sugestão aos desenvolvedores, um compilador baseado no método MCPON poderia ser dividido em dois programas distintos, separando efetivamente a camada de *front-end* da camada de *back-end*. Nesse âmbito, a integração destas duas camadas poderia ser realizada por meio dos grafos especializados otimizados, os quais poderiam ser disponibilizados em algum formato padronizado (*e.g.*, JSON ou XML), seja em um arquivo de texto ou em formato binário, para utilização destes como entrada para o *back-end* do compilador.

Em suma, a função do *back-end* consiste essencialmente em recuperar o grafo gerado no *front-end* e transformá-lo em um código-alvo em qualquer linguagem de programação ou, até mesmo, em linguagem de máquina, circuito eletrônico ou representações intermediárias outras. É importante observar que o *front-end* poderia ser totalmente desacoplado do *back-end*, uma vez que tais etapas utilizam como base uma representação intermediária comum (*i.e.*, Grafo PON), o que viabiliza a construção de uma base de compiladores, mais especificamente, geradores de código que poderiam ser reaproveitados em qualquer nova linguagem que gere grafos especializados de acordo com o Grafo PON. Isso se constitui em um sistema de compilação, conforme ilustra a Figura 54.

Figura 54: Sistema de compilação do PON



Fonte: Autoria Própria

Conforme apresenta a Figura 54, o sistema de compilação do PON, na camada de *front-end* pode ser composto por qualquer linguagem PON que contemple as características do paradigma. A camada intermediária ou *middle-end* é representada pelo Grafo PON (ou instâncias deste, na prática). A camada de *back-end*, por fim, poderia ser qualquer implementação/materialização do PON, incluindo os *Frameworks* e tecnologias outras implementadas à luz de sua teoria, conforme o Capítulo 3 ou, mais precisamente, a Seção 3.4. Isso possibilita uma integração de N linguagens distintas com M compiladores distintos, proporcionando uma integração entre estes, de modo que novas linguagens ou novos compiladores, aproveitariam todo um sistema completo de validações e otimizações comuns neste ambiente.

Finalmente, a fase de validação também poderia ser tanto integrada a um programa completo (com *front-end* e *back-end*) como também somente para o programa responsável pela síntese no processo de compilação. Em ambos os casos, a fase de validação apresentaria o objetivo de validar a integralidade dos geradores de código desenvolvidos. Em geral, o MCPON apresenta um desacoplamento entre as etapas, o que viabiliza a construção de ferramentas diversas com certa flexibilidade e facilidade, principalmente por ter como base o Grafo PON.

4.5 CONSIDERAÇÕES SOBRE O MÉTODO MCPON

Conforme apresentado ao longo deste capítulo, o método MCPON é fortemente baseado no Grafo PON. Estas inovações propostas neste trabalho, em conjunto, permitem que desenvolvedores possam construir todo o processo de compilação de cada linguagem específica para o PON, desde a definição desta linguagem até a geração de código para *targets* específicos e posterior validação do processo como um todo. Isto, de fato, norteia o desenvolvimento de compiladores próprios para o PON que sejam regidos por tal tecnologia.

É importante ressaltar que o Grafo PON se faz presente em todas as etapas do método MCPON. De maneira mais precisa, na etapa de definição de linguagem, o Grafo PON permite a concepção de linguagens regidas pelo PON, assim como também se intersecciona com a segunda etapa, responsável pela construção das instâncias dos grafos propriamente ditas. Com base nos grafos especializados construídos na segunda etapa é possível aplicar um conjunto de regras de otimização, as quais compõem essencialmente a terceira etapa do processo de compilação.

Ademais, a quarta etapa, responsável pela geração de código é fortemente baseada nos construtos do Grafo PON, gerados na etapa de otimização. Por fim, a última etapa, responsável pela validação dos *targets* gerados, também é fortemente integrada ao Grafo PON, principalmente no âmbito de validar programas em forma de grafos otimizados.

Em suma, o MCPON possibilita que todo o processo seja orientado por meio desta estrutura comum (*i.e.,* Grafo PON). De maneira geral, esse modelo uniforme para a construção de compiladores é uma inovação no processo de compilação para o PON, pois possibilita que sejam construídos programas-alvo para diferentes linguagens, paradigmas e plataformas, sem a necessidade de adaptar a estrutura do grafo gerado para plataformas específicas.

Outrossim, sejam os programas-alvo definidos para executar de forma monoprocessada, paralela ou distribuída, cada qual pode ser gerado a partir do mesmo grafo. Esse fato por si só, representa um avanço pertinente no âmbito de desenvolvimento de software, uma vez que criar programas que executam de forma paralela, ou até mesmo distribuída, é tida como uma tarefa relativamente complexa via linguagens e técnicas de programação usuais (conforme tratado na Seção 2.1). O PON, mais precisamente, por meio de sua linguagem e compilador, tornaria essa tarefa transparente ao desenvolvedor, reforçada pela propriedade elementar do paradigma que visa a facilidade de programação em alto nível orientado particularmente a regras.

Por fim, é possível concluir que o Grafo PON representa uma importante contribuição para o processo de compilação de linguagens específicas para o PON, trazendo inúmeras vantagens para o processo como um todo. Como algumas destas vantagens cita-se a possibilidade de criar linguagens distintas para o PON que se baseiam em uma mesma representação intermediária. Além disso, é possível criar geradores de código para plataformas distintas com base nesta mesma representação intermediária. Tais características, no final das contas, possibilitam a interligação de N para N, entre linguagens e compiladores distintos, proporcionando uma integração entre estes, de modo que novas linguagens ou novos compiladores, aproveitariam todo um sistema completo de validações e otimizações comuns neste ambiente.

Isto posto, os capítulos seguintes apresentam a implementação efetiva do método MCPON na construção de linguagens e compiladores próprios ao PON. De maneira sucinta, o método foi aplicado em sua versão preliminar (ou *alfa*), no âmbito

da criação da LingPON, conforme apresentado no Capítulo 5. O capítulo 6, por sua vez, apresenta estudos, no qual são contempladas e demonstradas todas as etapas do método proposto, principalmente na concepção da linguagem NOPL, também conhecida por LingPON 2.0 e seus compiladores associados.

CAPÍTULO 5

MATERIALIZAÇÃO DO MCPON PRELIMINAR: SISTEMA DE COMPILAÇÃO PRELIMINAR PARA TECNOLOGIA LINGPON

Este capítulo apresenta e avalia a materialização do método MCPON, ainda que em sua versão preliminar, particularmente com um viés de contextualização histórica. Este capítulo é importante, uma vez que o amadurecimento do método em si (apresentado em sua versão efetiva no Capítulo 6) foi fortemente amparado pelo acúmulo de experiências tanto individuais do autor desta tese, quanto coletivas no âmbito do grupo de pesquisa do PON. Para tal, este capítulo apresenta a criação da primeira linguagem de programação para o PON, denominada de LingPON e seu respectivo compilador próprio associado, ambos criados pelo autor desta tese. Tal linguagem e compilador deram origem a um conjunto de técnicas próprias ao PON que consequentemente deram origem ao próprio método MCPON e sua implementação em forma de um Sistema de Compilação Preliminar.

Objetivamente, o Sistema de Compilação Preliminar é a implementação do MCPON Preliminar, por meio de um conjunto de tecnologias, que permitem a criação de uma ou mais linguagens, bem como um ou mais compiladores para o PON. Importante destacar que neste momento da elaboração da tese já se considerava o Sistema de Compilação Preliminar, enquanto que o MCPON Preliminar era tido como um proto-método explicado de forma textual estruturada e tutoreada na sua utilização no seio do grupo de pesquisa do PON.

Além disso, o capítulo apresenta as evoluções das tecnologias desenvolvidas pelos pesquisadores e colaboradores do grupo de pesquisa PON, baseados nas diretrizes da versão preliminar do método MCPON e, naturalmente do Sistema de Compilação Preliminar em seus trabalhos[11]. Certamente, tais desenvolvimentos serviram basicamente para validar tanto o MCPON Preliminar quanto o seu Sistema de Compilação Preliminar, havendo trabalhos em diferentes etapas do método, a

[11] Em tempo, é pertinente salientar que, a partir do MCPON preliminar e respectivo sistema de compilação associado, as evoluções na LingPON e a construção de demais compiladores/geradores de código foram, em sua maioria, acompanhados pelo autor desta tese, primeiramente em coautoria e depois tutoreando os demais pesquisadores e colaboradores na aplicação de tal tecnologia.

maioria em expansão da linguagem e/ou geradores de código com validação por experimentação individual.

Nesse âmbito, primeiramente, a Seção 5.1 apresenta o Sistema de Compilação Preliminar, a decorrente LingPON prototipal com suas particularidades e os subsequentes primeiros três geradores de códigos, os quais, em conjunto, permitiram compor os três primeiros compiladores à luz do MCPON Preliminar. Este conjunto de tecnologias foi denominado de Tecnologia LingPON Prototipal. A Seção 5.2, por sua vez, apresenta evoluções linguísticas que foram chamadas de LingPON 1.0, suas características e respectiva atualização de compiladores, no tocante a geração de código apenas, rebatizando tudo como Tecnologia LingPON 1.0. A Seção 5.3 mostra as evoluções no Sistema de Compilação Preliminar em si por alguma expansão do Grafo PON Preliminar, alguma expansão da LingPON em termos de recursos linguísticos e dos geradores de código (permitindo alcançar compiladores distintos, mas de mesma base MCPON) ao longo dos últimos anos, condensadas na chamada Tecnologia LingPON 1.2.[12]

Particularmente, a Seção 5.4 apresenta um dialeto da linguagem e compilador para Hardware Digital, denominado de LingPON HD, relatando versão independente do MCPON e outra subsequente integrada ao MCPON, o que foi finalmente denominado como Tecnologia LingPON HD. A Seção 5.5 apresenta outro dialeto da linguagem, com alguma evolução correlata do Grafo PON Preliminar, voltada a aplicações baseadas em lógica *fuzzy* chamada Tecnologia LingPON Fuzzy. A Seção 5.6, por sua vez, apresenta uma versão experimental da LingPON e gerador de código para execução paralelizada, com base em técnicas de *multithreading*, o que pode ser considerado como pertencente a Tecnologia LingPON 1.2. Por fim, a Seção 5.7 apresenta as considerações sobre este capítulo, elencando os pontos fortes e pontos fracos do método em sua versão preliminar, bem como em sua aplicação prático-experimental. Ademais, esta seção destaca a pertinência das evoluções apresentadas e o impacto destas no alcance de materializações efetivas para o PON.

[12] Pertinente ressaltar que todas essas seções citadas, mais a seção 5.5 subsequente, foram melhoradas a partir da qualificação de doutorado do autor desta tese (RONSZCKA, 2018), a qual ensejou esse presente trabalho. Uma principal diferença, neste presente texto de trabalho de doutorado, é a explicação desses desenvolvimentos explicitamente à luz do MCPON Preliminar.

5.1 TECNOLOGIA LINGPON – VERSÃO PROTOTIPAL

Os estudos relacionados a construção de uma linguagem e compilador específicos para o PON começaram a surgir no grupo de pesquisa em questão no ano de 2013, cf. relatório técnico pertinente (RONSZCKA *et al.,* 2013), por desejo do grupo de pesquisa do PON[13]. Em suma, esse esforço inicial permitiu evidenciar a viabilidade do desenvolvimento de uma linguagem específica e de sistema de compilação, levando a alcançar um compilador próprio ao PON, à luz de técnicas de compilação, sendo parte delas tradicionais e outra parte já própria para o PON.

De maneira geral, a especificação formal da LingPON prototipal teve sua gramática especificada segundo a *Backus-Naur Form* (*BNF*), à luz da teoria usual de linguagens de programação descrita na Seção 2.2.1. A linguagem em si é fortemente baseada nos princípios do PON, tendo a expressão dos elementos do modelo definida de forma clara e explícita, de modo a apresentar uma linguagem de compreensão intuitiva.

O Sistema de Compilação Preliminar para o LingPON em questão, por sua vez, foi construído usando tecnologias tradicionais de compiladores, nomeadamente Flex/Lex e Bison (conforme Seção 4.3.1), no tocante às análises léxica e sintática principalmente, sendo que estes geram módulos de análise integráveis com códigos C/C++ que integram o restante do sistema. Entretanto, para permitir integrar tais módulos com a estrutura léxica-sintática da LingPON prototipal, com a parte geração de código desenvolvida, não foi possível usar as usais árvores sintáticas e tabelas de símbolos (sem alterações) da teoria de compiladores devido à natureza do PON.

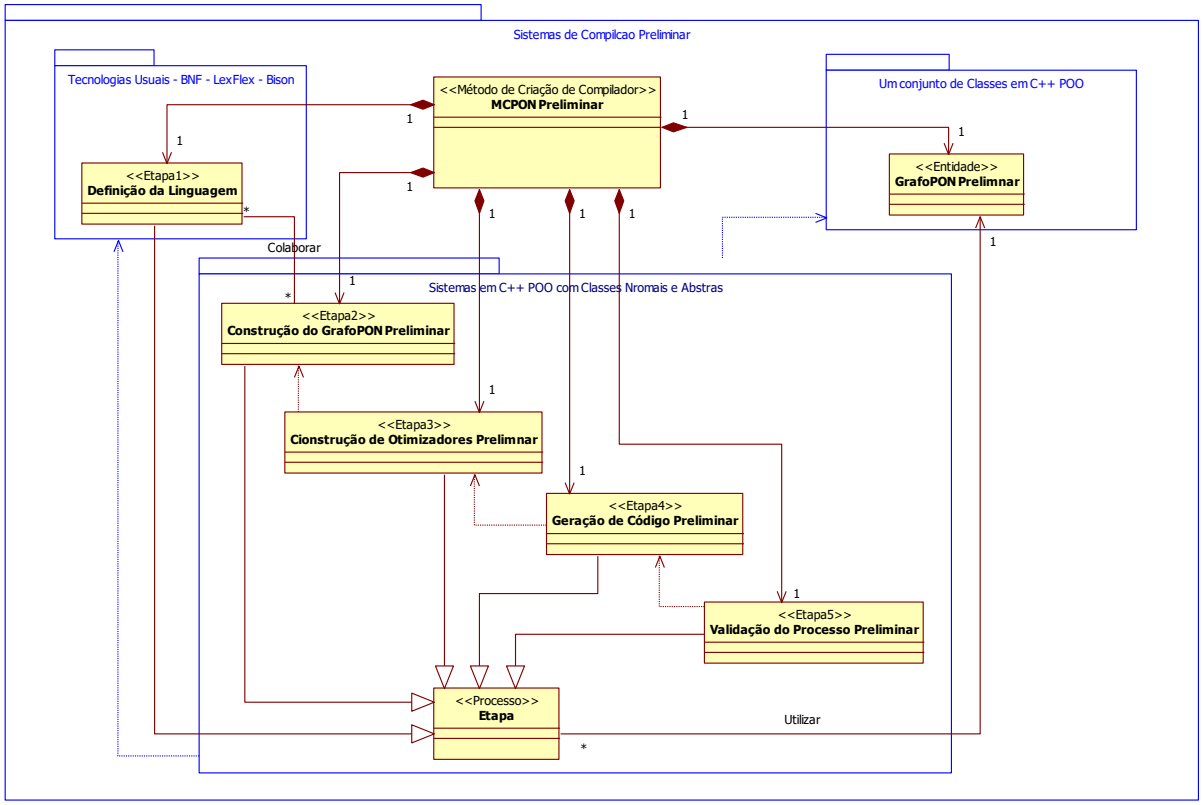
Conforme apresentado no Capítulo 4, mais precisamente nas seções 4.1 e 4.2, as técnicas de compilação utilizadas nos paradigmas usuais não são apropriadas para o PON, principalmente por se apoiarem em um mapeamento sequencial e orientado a elementos passivos, representado na forma de árvores de aninhamentos

---

[13] Em um primeiro momento, tais estudos surgiram no âmbito de uma disciplina de Linguagens e Compiladores, ofertada na UTFPR em meados de 2013 (2º trimestre) com aquele propósito em mente (*i.e.,* de iniciar pesquisas em linguagens e compiladores para o PON), a qual foi ministrada pelos professores J. A. Fabro e J. M. Simão. Naturalmente, após o impulso inicial dado pela disciplina, os estudos se estenderam no seio do grupo de pesquisa, resultando primeiramente na chamada Tecnologia LingPON Prototipal e subsequentemente na Tecnologia LingPON 1.0, tudo já à luz de do MCPON Preliminar e de respectivo Sistema de Compilação Preliminar definidos por A. F. Ronszcka. Em tempo, os pesquisadores discentes que participaram do esforço inicial no âmbito da Tecnologia LingPON Prototipal foram A. F. Ronszcka, C. A. Ferreira, C. Kossoski e P. A. de Moraes Ioris.

de comandos e instruções. No PON, idealmente, o mapeamento de um programa deveria se dar por meio de um grafo de entidades conectadas por meio de suas notificações. Isto já se observou de pronto no primeiro protótipo do Sistema de Compilação Preliminar, o que exigiu o advento de um novo conceito e uma nova técnica de compilação decorrente, o que veio a ser chamado de Grafo PON. Em tempo, a Figura 55 retrata com quais tecnologias o Sistema de Compilação Preliminar foi implementado.

Figura 55: Tecnologias usadas no Sistema de Compilação Preliminar



Fonte: Autoria Própria

Basicamente, mesmo que em versão preliminar, o Grafo PON é um mapeamento fidedigno da estrutura de um programa PON em forma de grafo. Tal grafo é composto pelas instâncias especializadas das entidades do PON e suas conexões baseadas em notificações, bem como por outras entidades auxiliares. Com base em tal grafo, a etapa de tradução de código se torna desacoplada de todas as etapas anteriores do processo.

Na verdade, o Grafo PON é essencialmente uma representação intermediária independente de *target* que permite a associação de linguagens distintas com geradores de código distintos. Isto, associado com as fases anteriores que alimentam

o Grafo PON e as fases dele decorrentes, instituem assim um Sistema de Compilação Preliminar, o qual devidamente estruturado permitiu emergir o método MCPON (primeiramente *Preliminar* e depois também *Efetivo*). Tal sistema de compilação, à luz da estruturação do método, permite a criação de vários desdobramentos de linguagens e compiladores, possibilitando a criação de uma base de *N* linguagens e *M* geradores de código independentes entre si.

Com base nisso, já de início o Sistema de Compilação Preliminar do PON, à luz do Grafo PON também prototipal, contou com três versões iniciais de geração de código, para linguagens-alvo distintas, que são:

(a) compilação para *Framework* PON C++ 2.0 com o intuito de simplificar o processo de criação de programas para essa ferramenta – o que é chamado de versão Framework PON C++ 2.0;
(b) compilação para uma versão em linguagem C, em um estilo procedimental, na qual as notificações são representadas por chamadas de funções em um arranjo próprio ao PON – o que é chamado de versão C específica a notificações; e
(c) compilação para uma versão em linguagem C++, orientada a objetos, simplificando a estrutura de classes do Framework PON C++ 2.0 e amenizando o gargalo de desempenho no processo de notificação que ocorre quando se usa o framework em questão – o que é chamado de versão C++ específica a notificações.

Nesse âmbito, a implementação de três geradores de código distintos possibilitou tirar conclusões importantes sobre a importância de uma linguagem própria e compilador eficiente para o PON. De maneira geral, os resultados se mostraram bastante promissores, evidenciando o potencial da compilação de programas regidos pela essência do PON. Tais resultados foram registrados em relatório técnico, cf. (RONSZCKA *et al.,* 2013) e são considerados subsequentemente em subseção desta presente seção.

Ademais, outra vantagem que a linguagem trouxe em relação às materializações precedentes foi a facilidade de implementação de programas por meio de um modelo em alto nível, o qual se apresentou efetivamente orientado a regras. Isso é diferente das implementações baseadas em *frameworks* principalmente

porque, não obstante a existência de um conjunto de facilitadores como interfaces, eles ainda carregam o peso da curva de aprendizado das linguagens em que foram implementadas, as quais exigem conhecimento específico.

Nesse âmbito, de modo a apresentar os detalhes desse esforço inicial, a Tabela 6 apresenta a relação das atividades realizadas na criação da chamada Tecnologia LingPON Preliminar, atividades naturalmente relativas às etapas do método MCPON Preliminar, o que evolve a criação do Sistema de Compilação Preliminar (com seu Grafo PON Preliminar), da LingPON em si e dos três geradores de código permitindo alcançar compiladores citados. Em tempo, além das atividades em si, a título de elucidações de autorias no tocante a cada pesquisador discente envolvido, também se traz na tabela os autores das principais atividades e colaboradores discentes em cada atividade.

Tabela 6: Atividades realizadas na criação da Tecnologia LingPON Prototipal

Etapa MCPON Preliminar	**Atividades relacionadas as Etapas do MCPON Preliminar, visando o Sistemas de Compilação Preliminar, alcançando a Tecnologia LINGPON Prototipal.**	**Autor principal e colaborador(es)**
Etapa 0.	Criação do Grafo PON Preliminar.	Adriano F. Ronszcka
Etapa 1.	Elaboração da LingPON – Gramática *BNF*.	Adriano F. Ronszcka
Etapa 1 & Etapa 2.	Elaboração do Sistema de Compilação Preliminar que usa (popula, fornece acesso etc.) o Grafo PON preliminar.	Adriano F. Ronszcka.
Etapa 4.	Elaboração de Gerador de Código para *Framework* PON C++ 2.0, permitindo conjuntamente com as outras atividades e etapas anteriores alcançar um primeiro compilador PON (**versão Framework PON C++ 2.0**).	Adriano F. Ronszcka
Etapa 4.	Elaboração de Gerador de Código para implementação em linguagem C++ específica (com orientação) a notificações por objetos notificantes concretos, permitindo conjuntamente com as outras atividades e etapas (já prontas) anteriores alcançar um segundo compilador PON (**versão C++ *específica a notificações***).	Cleverson A. Ferreira, Adriano F. Ronszcka.
Etapa 4.	Elaboração de Gerador de Código para implementação em linguagem C específica (com orientação) a notificações por chamadas de função, permitindo conjuntamente com as outras atividades e etapas anteriores (já prontas) alcançar	Priscila A. M. Ioris, Adriano F. Ronszcka.

	um terceiro compilador PON (**versão C *específica a notificações***).	
Etapa 5.	Testes preliminares de funcionamento, desempenho e consumo de memória da Tecnologia LingPON por meio de programa exemplo (*toy* problem – mira alvo).	Clayton Kossoski, Adriano F. Ronszcka.

Fonte: Autoria Própria

Conforme apresentado na Tabela 6, esse esforço inicial possibilitou criar a linguagem LingPON prototipal, o Sistema de Compilação Preliminar para o PON com o Grafo PON Preliminar. Com base nessa estrutura deste ferramental, foi possível criar três versões de geração de código e, portanto, três compiladores finalmente. Nesse âmbito, de maneira a apresentar de forma pontual as características da Tecnologia LingPON Preliminar, as subseções seguintes do presente trabalho apresentam em detalhes a linguagem LingPON prototipal e o Sistema de Compilação Preliminar desenvolvidos[14].

5.1.1 Linguagem de programação LingPON Prototipal

De maneira geral, a especificação da linguagem de programação LingPON Prototipal segue uma estrutura de programação declarativa. Essencialmente, um programa em PON é definido em três seções principais, que são: (a) Definição de classes de *FBEs*; (b) Instanciações de *FBEs*; e (c) Declaração de *Rules* (RONSZCKA *et al*., 2017b). O Código 18 apresenta a organização de um código escrito em LingPON, bem como as palavras reservadas para definição de cada seção.

[14] As descrições da Tecnologia LingPON Preliminar se encontram nos seguintes trabalhos: (RONSZCKA *et al.,* 2013), (FERREIRA, 2015), (RONSZCKA *et al.,* 2017b) e (RONSZCKA, 2018), sendo este último a qualificação de doutorado do autor desta tese. Entretanto, neste documento de tese optou-se por fazer um relato mais estruturado e compreensível do que os trabalhos anteriores, particularmente à luz do MCPON Preliminar e do Sistema de Compilação Preliminar, salientado que isso não ocorria a contento naqueles trabalhos.

Código 18: Estrutura base de declarações da LingPON

```
1  fbe FbeName
2  . . .
3  end_fbe
4
5  inst
6  . . .
7  end_inst
8
9  rule ruleName
10 . . .
11 end_rule
```

Fonte: Autoria Própria

Conforme apresenta o Código 18, é possível observar que a primeira seção da estrutura de um programa em LingPON permite a definição de estruturas de *FBEs*. A palavra reservada ***fbe*** anuncia o início da definição de um *FBE* e a palavra ***end_fbe***, por sua vez, finaliza tal definição. Nessa seção, o desenvolvedor pode declarar livremente quantos *FBEs* forem necessárias para estruturar seu programa.

A segunda seção consiste na criação de um bloco de instâncias de *FBEs*, no qual o desenvolvedor pode criar instâncias indiscriminadamente de cada um dos *FBEs* definidos na seção anterior. O bloco de instâncias é iniciado por meio da palavra reservada ***inst***, sendo que todas as declarações nesse bloco devem representar exclusivamente instâncias dos tipos conhecidos de *FBEs*. Uma vez que o bloco esteja definido, a utilização da palavra ***end_inst*** se faz necessária para efetivar a construção das instâncias do programa em questão.

Por fim, a terceira seção é designada para a declaração das *Rules* do programa. A palavra reservada ***rule*** anuncia a declaração de uma *Rule* particular, a qual tem como objetivo principal organizar e conectar todas as entidades notificantes pertinentes, em prol de estruturar uma expressão lógico-causal, bem como definir a ação a ser executada na aprovação desta *Rule*. Assim como os demais blocos possuem uma palavra reservada para abertura e fechamento de suas definições, a *Rule* em si é concretizada a partir da palavra reservada ***end_rule***.

Cabe ressaltar que a primeira seção (*i.e.,* Definição de classes de *FBEs*) e a terceira seção (*i.e.,* Declaração de *Rules*) não possuem um início e fim de bloco geral. As definições e declarações são livres no código, possibilitando a criação de um número ilimitado de *FBEs* e *Rules* em suas respectivas seções.

A título de exemplo da primeira seção, na qual são definidos os *FBEs* do programa, o Código 19 traz a criação de um *FBE*. Conforme apresenta o Código 19,

a criação de um FBE deve possuir dois blocos, um para definição dos *Attributes* e outro para definição dos *Methods*.

Código 19: Exemplo de estrutura de um FBE na LingPON

```
1  fbe Alarm
2    attributes
3      boolean atOn false
4      integer atTimer 0
5    end_attributes
6    methods
7      method mtRingTheBell(atTimer = 60)
8    end_methods
9  end_fbe
```

Fonte: Autoria Própria

Na definição de cada *FBE*, o bloco de *Attributes* é definido pelas palavras reservadas ***attributes*** e ***end_attributes***. Os *Attributes* possuem uma estrutura comumente utilizada na programação, formada pelo tipo do dado, seguido do nome do Attribute e seu respectivo valor inicial. Nesse âmbito, os tipos de dados disponíveis na LingPON preliminar são ***boolean***, ***integer***, ***float***, ***char*** e ***string***.

O bloco de *Methods*, por sua vez, é definido pelas palavras reservadas ***methods*** e ***end_methods***. Cada *Method* definido nesse bloco deve ser anunciado pela palavra reservada ***method***, seguido de um nome único e sua respectiva função, a qual deve estar entre parênteses. A função de um *Method* pode ser tanto uma atribuição simples de um valor constante, assim como uma expressão aritmética, resultando em um novo valor para um dos *Attributes* do respectivo *FBE*.

A segunda seção de programa definida na LingPON, particularmente, consiste na criação de um bloco de instâncias dos *FBEs* definidos na primeira seção. Para melhor compreensão, o Código 20 exemplifica o padrão de implementação desse bloco.

Código 20: Exemplo de bloco de instâncias de *FBEs* na LingPON

```
1  inst
2    Alarm alarm1
3    SensorTemp sensorTemp1
4    SensorPresence sensorPresence1, sensorPresence2
5  end_inst
```

Fonte: Autoria Própria

Conforme apresenta o Código 20, as palavras reservadas ***inst*** e ***end_inst*** anunciam o bloco de instanciações de *FBEs*. Essencialmente, nesse bloco, o desenvolvedor precisa informar o respectivo nome da definição do *FBE* a ser

instanciado, bem como o nome da instância a ser criada. Assim como exemplificado no Código 20, mais especificamente na linha 4, é possível criar duas ou mais instâncias do mesmo *FBE*, separando cada nova instância por vírgulas.

Por fim, na terceira seção de um programa PON, o desenvolvedor deve expressar o conhecimento causal de seu programa na forma de *Rules*. Assim como mencionado anteriormente, a linguagem foi estruturada de forma clara e explícita, de modo a apresentar uma essência particularmente didática. Nesse âmbito, boa parte dos elementos conceituais do PON são apresentados na construção de uma *Rule*.

O Código 21 exemplifica a criação de uma *Rule* para o caso de estudo aplicado. Neste código, uma *Rule* é composta por duas partes principais, que são suas respectivas *Condition* (expressão lógica-causal) e *Action* (execução). O bloco de *Condition* é definido pelas palavras reservadas ***condition*** e ***end_condition***. Cada *Rule* pode possuir apenas um bloco de *Condition*, porém, é possível definir mais de uma *SubCondition* para compor *Rules* mais elaboradas.

Código 21: Exemplo de criação de *Rule* na LingPON

```
1  rule rlFireAlarm
2    condition
3      subcondition scAlarmOn
4        premise prAlarmOn alarm1.atOn == true
5      end_condition
6      and
7      subcondition scIntruderIdentified
8        premise prSensorPresence1 sensorPresence1.atStatus == true
9        or
10       premise prSensorPresence2 sensorPresence2.atStatus == true
11     end_subcondition
12   end_condition
13   action
14     instigation inFireAlarm alarm1.mtRingTheBell();
15   end_action
16 end_rule
```

Fonte: Autoria Própria

No exemplo, as linhas 3 a 5 e 7 a 11 ilustram dois blocos de subcondições anunciados pelas palavras reservadas ***subcondition*** e ***end_subcondition***. Uma *SubCondition* pode conter um nome único (opcional) com o objetivo de simplificar a leitura das *Rules* do programa. Ainda, no caso de o desenvolvedor optar por criar mais de uma *SubCondition*, essas precisam estar separadas por uma conjunção ou disjunção, representadas respectivamente pelas palavras reservadas ***and*** ou ***or***.

Ademais, cada *SubCondition* precisa ser composta por pelo menos uma *Premise*, as quais representam as relações causais entre *Attributes* e um estado

esperado para esse. Cada *Premise* definida para uma *SubCondition* deve ser anunciada pela palavra reservada ***premise***, seguida de um nome único e sua respectiva comparação. A comparação da *Premise* apresenta três partes: (a) um *Attribute* de uma instância particular de um *FBE* (*e.g. Atttribute atStatus* da instância *sensorPresence1* do *FBE SensorPresence*), o valor de comparação (*e.g.* ==) e o valor a ser comparado (*e.g. true*). Esse último pode ser tanto uma constante, quanto outro *Attribute*. Ademais, os operadores de comparação possíveis para uma *Premise* são: ==, <, >, <=, >= e !=. Ainda, para subcondições com mais de uma *Premise*, as palavras reservadas ***and*** ou ***or*** devem ser utilizadas para conectar os resultados das avaliações lógicas de cada *Premise*, formando assim uma expressão causal completa.

A segunda parte, que representa a execução de uma *Rule*, é anunciada por meio das palavras reservadas ***action*** e ***end_action***. Esse bloco consiste no vínculo entre *Instigations* de uma *Action* e *Methods* de um *FBE*. Assim como a *SubCondition*, a *Action* da *Rule* pode conter um nome único (opcional) com o objetivo de simplificar a leitura da mesma. Essencialmente, uma *Action* pode ser composta por uma ou mais *Instigations* de *Methods*. Conforme apresentado no Código 21, mais precisamente na linha 14, uma *Instigation* é anunciada pela palavra reservada ***instigation*** e deve ser seguida de um nome obrigatório e uma única chamada de *Method* de uma das instâncias de *FBEs* declaradas no programa.

É importante salientar que todas as instâncias de *FBEs* estão acomodadas em um escopo global, possibilitando a criação de *links* nas *Rules* entre todos os elementos de forma livre, sem restrições de escopo (RONSZCKA *et al.,* 2017b).

5.1.2 Sistema de Compilação Preliminar para o PON

De maneira geral, o sistema de compilação para o PON foi construído seguindo as diretrizes do método MCPON em sua versão preliminar, apresentando um *front-end* com as principais etapas de análise (*i.e.,* léxica e sintática da Etapa 1 do MCPON Preliminar), um *middle-end* baseado na representação intermediária Grafo PON preliminar (Etapa 2 do MCPON Preliminar) e um *back-end* composto essencialmente pela etapa de geração de código (Etapa 4 do MCPON Preliminar). É importante ressaltar que essa primeira versão do Sistema de Compilação Preliminar foi construída de maneira um tanto simplificada, com base apenas nas principais subetapas das etapas do MCPON Preliminar. Nesse âmbito, as próximas subseções

detalham as etapas e subetapas em questão que levaram a composição do Sistema de Compilação em pauta.

5.1.2.1 *Front-end* – Etapa 1 do MCPON Preliminar.

O Sistema de Compilação Preliminar do PON foi naturalmente construído contemplando as análises léxica e sintática no âmbito do *front-end* deste sistema, i.e., no tocante Etapa 1. Em termos de linguagem, o *front-end* desta versão foi totalmente baseado na primeira SubEtapa da Etapa 1 (Definição da linguagem). Assim sendo, contemplou-se todas as subetapas da Etapa 1 do método que são: (a) Definição das Características da Linguagem; (b) Definição das Palavras-chave e Analisador Léxico; e (c) Definição das Regras Gramaticais e Analisador Sintático.

Isto posto, o analisador léxico foi construído com base na ferramenta *open-source* Flex. Em tal ferramenta foram definidas todas as palavras reservadas pertinentes a LingPON, bem como a identificação de IDs, números e demais elementos para uma correta interpretação dos *tokens* da linguagem. Nesse âmbito, algumas das regras léxicas da LingPON são apresentadas no Código 22.

Código 22: Exemplo de regras léxicas da LingPON preliminar

```
1  Expressão Regular            -> TOKEN
2
3  [0-9]+                       -> NUMBER
4  [a-zA-Z\_]([a-zA-Z0-9\_])*   -> ID
5
6  rule                         -> RULE
7  condition                    -> CONDITION
8  action                       -> ACTION
9  instigation                  -> INSTIGATION
10 premise                      -> PREMISE
11 attribute                    -> ATTRIBUTE
12 method                       -> METHOD
13 fbe                          -> FBE
14
15 integer                      -> INTEGER
16 boolean                      -> BOOLEAN
17 float                        -> FLOAT
18 string                       -> STRING
19 char                         -> CHAR
```

Fonte: Autoria Própria

Conforme apresenta o Código 22, as regras léxicas da LingPON são baseadas em expressões regulares, como as utilizadas para o mapeamento de números (linha 3) e IDs (linha 4). Além disso são apresentadas algumas das palavras

reservadas (linhas 6 a 13), bem como os tipos de dados dos *Attributes* (linhas 15 a 19).

A análise sintática, por sua vez, foi desenvolvida a partir da ferramenta *open-source* Bison (BISON, 2019), a qual já possui uma integração com a ferramenta Flex (FLEX, 2019), possibilitando interpretar os *tokens* extraídos pela etapa anterior e mapeá-los na análise sintática de acordo com a *BNF* da linguagem especificada na ferramenta Bison. De modo a apresentar a relação entre as duas ferramentas, o Código 23 apresenta algumas das regras gramáticas definidas na LingPON.

Código 23: Exemplos de regras gramaticais da BNF da LingPON preliminar

```
1  <program>              ::= <fbes> <inst> <rules>
2
3  <fbes>                 ::= <fbe>
4                           |  <fbe> <fbes>
5
6  <fbe>                  ::= FBE <id> <fbe_body> END_FBE
7                           |  FBE <fbe_body> END_FBE
8
9  <fbe_body>             ::= <decl_attributes> <decl_methods>
```

Fonte: Autoria Própria

O Código 23 apresenta um trecho da BNF da LingPON preliminar, cuja BNF completa é apresentada no Apêndice B. Nesse trecho, em particular, são apresentados os três blocos principais da linguagem. Mais precisamente, a linha 1 apresenta o símbolo inicial (*i.e.,* <*program*>), o qual basicamente define que a estrutura de um programa é composta por três símbolos não-terminais principais (*i.e.,* <*fbes*>, <*inst*> e <*rules*>). Para cada um desses três símbolos não-terminais existe uma produção gramatical que define a forma com que eles são descritos no código-fonte. Basicamente, nas linhas 3 e 4 as produções definem que o bloco de definições de classes de *FBE* podem ter um número ilimitado de ocorrências. As linhas 6 e 7, por sua vez, expressam definições de classes de *FBEs*, as quais podem ser opcionalmente identificadas por um nome (*i.e.,* <*id*>). Por fim, a linha 9 expressa que o corpo de uma classe de *FBE* é composta por declarações de *Attributes* e de *Methods*.

Com base nas ferramentas Flex e Bison, os quais geram módulos com analisadores léxico e sintático em linguagem C, é possível então criar a integração da fase de análise (*front-end*), com a construção do Grafo PON (*middle-end*) que, em termos da materialização, se constitui em uma classe em C++ (com respetiva instância ou objeto) com os dados em forma de grafo e os métodos correlatos para popular os

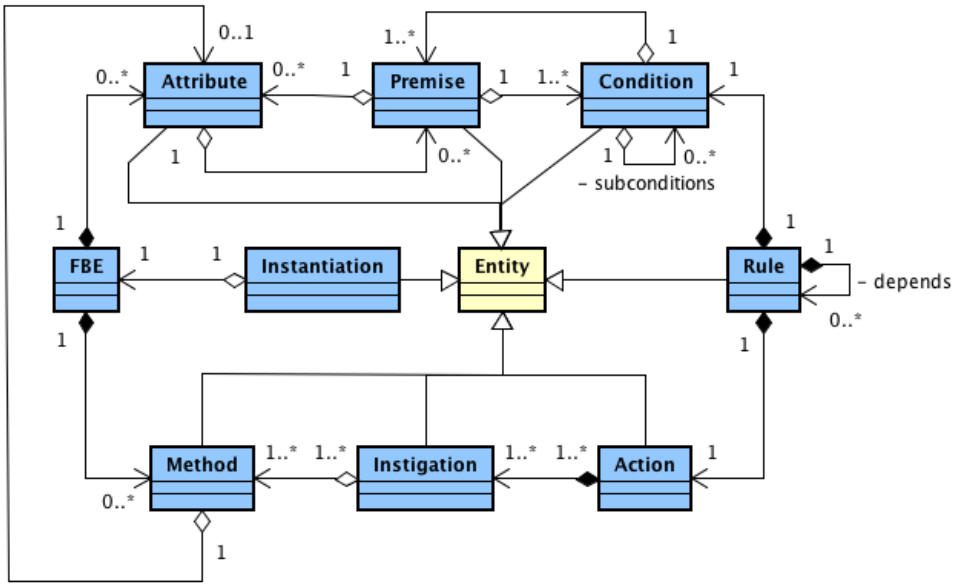
dados do grafo em si, acessá-los, modificá-los etc, conforme apresenta a próxima subseção.

5.1.2.2 *Middle-end* - Grafo PON

Normalmente o *front-end* de um compilador, responsável pela fase de análises (léxica, sintática e semântica) de um programa, é integrado ao *middle-end*, o qual é responsável por construir uma representação intermediária do programa analisado. Conforme já explicado, no caso da tecnologia LingPON, tal representação intermediária se dá na forma do Grafo PON que, na prática, poderia ser uma instância (*i.e.,* um objeto) de uma classe que permite agregar uma coleção de outras instâncias úteis de outras classes.

Oportunamente, é importante salientar que nessa versão preliminar do Sistema de Compilação também foi previamente criada uma versão igualmente preliminar do Grafo PON, a qual é apresentada em um diagrama de classes em UML na Figura 56. A estrutura de classes de entidades do Grafo PON Preliminar é representada essencialmente pelas entidades do paradigma com a adição de algumas entidades e conexões auxiliares. Essa versão do Grafo PON serviu como um primeiro artefato para verificar particularmente a proposição do Grafo PON como o elemento central do método MCPON, isso tanto nesta versão preliminar quanto na efetiva.

Figura 56: Grafo PON prototipal

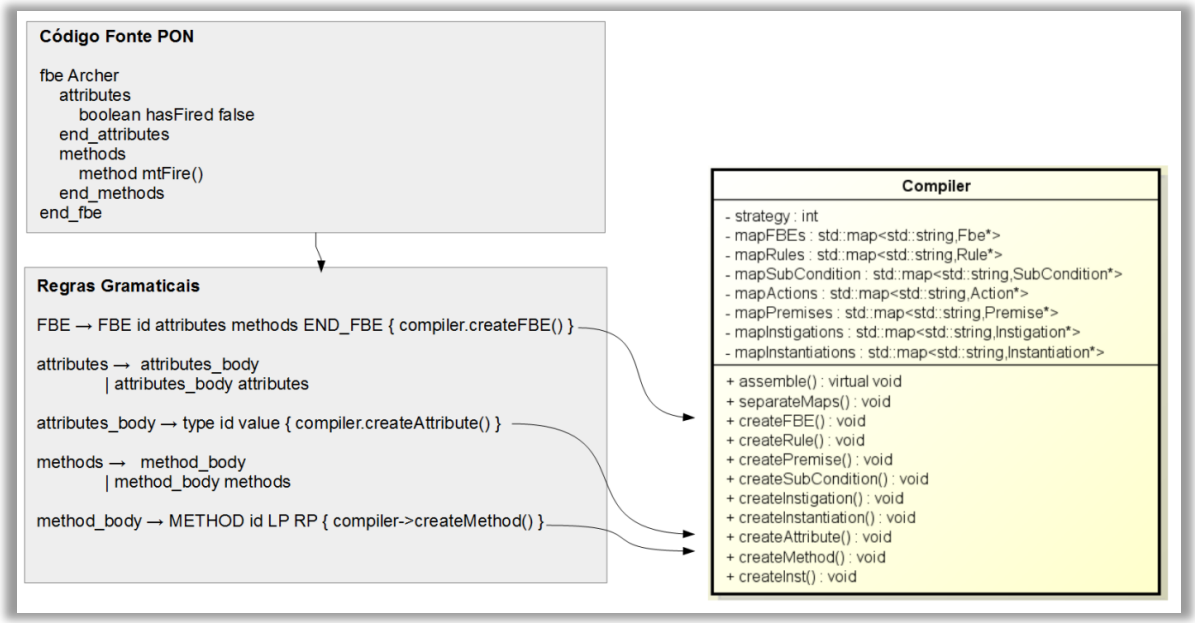


Fonte: Autoria Própria via Diagrama de classes em UML

Ainda, em termos de MCPON, a instanciação do Grafo PON e sua articulação enquanto *Middle-end* na forma de uma instância principal e um conjunto de instâncias outras faz parte da Etapa2. Neste caso, contemplou-se as seguintes subetapas: (a) instanciar as entidades e popular instâncias do Grafo PON; e (b) construir a integração das análises com o Grafo PON. Em tempo, não foi contemplada a última subetapa: (c) definição das regras semânticas e analisador semântico.

De modo a explicitar como é realizada a integração do processo de análise (léxico-sintática) com a construção do Grafo PON preliminar, a Figura 57 apresenta um exemplo do processo de execução das regras gramaticais para a extração, criação e armazenamento das entidades em uma instância do Grafo PON.

Figura 57: Exemplo do processo de execução de regras gramaticais



Fonte: Ferreira, 2015 no tocante a Tecnologia LingPON Prototipal

Conforme apresenta a Figura 57, o código fonte escrito em LingPON passa pela etapa de análise sintática, na qual a cada *match* nas regras gramaticais definidas na *BNF* da linguagem, são realizadas chamadas a métodos pontuais, os quais manipulam a construção do Grafo PON, armazenando os dados extraídos nas entidades pertinentes. Nesse exemplo, em particular, são realizadas chamadas aos métodos *createFBE*, *createAttribute* e *createMethod*, os quais são responsáveis pela criação e armazenamento das entidades no grafo.

Diferentemente da versão final do Grafo PON Efetivo ou Final (cf. Seção 4.2), nessa versão preliminar as entidades *FBEs* eram mapeadas em forma de classes,

deixando a cargo da etapa de geração de código a interpretação e instanciação de cada entidade conforme necessidade do *target* de compilação.

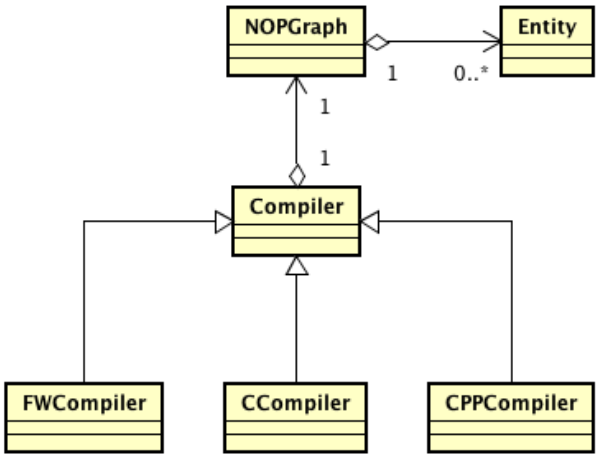
A hipótese para ter feito essa primeira versão desta maneira era ter uma representação intermediária mais genérica e, portanto, supostamente mais concisa dos entes pertinentes ao PON no tocante a construção de um sistema de compilação uniforme. Entretanto, essa hipótese não foi confirmada como se supunha, conforme será explicado depois de apresentar das materializações que ajudaram a testar essa proposição do Grafo PON.

5.1.2.3 *Back-end* – Gerador de Código

Em termos de MCPON, a geração de código ou *back-end* é contemplado na Etapa 4 (Geração de Código), nas suas duas subetapas: (a) iterar instâncias do Grafo PON; e (b) Construção de Geradores de Código. Em tempo, a etapa 3 (Construção de Otimizadores) não foi contemplada na Tecnologia LingPON prototipal.

Em suma, com base em instâncias do Grafo PON, o *back-end* está apto para gerar código-alvo para a linguagem-alvo definida. Nesse âmbito, de maneira a apresentar a estrutura resumida do sistema de compilação preliminar, a Figura 58 apresenta o diagrama de classes da estrutura de geração de código, bem como sua relação com as demais classes do modelo.

Figura 58: Estrutura de classes do Sistema de Compilação Preliminar



Fonte: Autoria Própria via Diagrama de classes em UML

Conforme apresenta o diagrama de classes da Figura 58, a classe *NOPGraph* se apresenta como uma interface para criação de instâncias do Grafo PON, que além disso também armazena todas as entidades instanciadas no processo de análise. A classe *Compiler,* basicamente, representa a estrutura do sistema de compilação como um todo, encapsulando todas as etapas do processo de compilação de um programa LingPON.

Ainda, com base nas diretrizes do método MCPON Preliminar, a estrutura de classes do sistema de compilação foi construída de forma a unificar a etapa de análise e de síntese. Para isso, é realizada inicialmente a extração das entidades pertinentes, armazenando-as apropriadamente de acordo com as diretrizes do Grafo PON, para posteriormente possibilitar a criação de diferentes geradores de código, por meio do mecanismo de herança.

Conforme apresentado na Figura 58, o Sistema de Compilação Preliminar foi inicialmente composto por três geradores de código distintos, representados pelas classes *FWCompiler*, *CCompiler* e *CPPCompiler*, cada qual com suas nuances pertinentes para geração de código-alvo particular. Nesse âmbito, as subseções subsequentes apresentam as particularidades de cada um dos geradores de código criados.

5.1.2.3.1 Geração de código para *Framework* PON C++ 2.0

O primeiro gerador de código criado para o Sistema de Compilação Preliminar do MCPON teve o propósito de testar a factibilidade da linguagem LingPON em relação à estrutura fundamental do paradigma, uma vez que ela transforma as entidades do Grafo PON em entidades correspondentes na estrutura do *Framework* PON C++ 2.0. A escolha do *framework* para esse fim foi definida pelo fato de que este já apresentava apropriado grau de maturidade em materializar o estado da arte do paradigma em um programa executável em arquitetura Von Neumann monoprocessado (RONSZCKA *et al*., 2017a; SIMÃO *et al*., 2017c).

Na prática, o gerador de código para *framework* basicamente instancia o conjunto de classes do *Framework* PON C++ 2.0 de acordo com o grafo PON gerado na etapa anterior. Além disso, o gerador cria todos os arquivos (*i.e., .h* e *.cpp*) pertinentes, com toda a estrutura do código preparada para simplesmente compilar

no *GCC* com a estrutura do *framework*. Nesse âmbito, o Código 24 e o Código 25 apresentam o código gerado para um *FBE* em particular.

Código 24: Código gerado para as declarações do arquivo Alarm.h

```
1  #ifndef Alarm_H_
2  #define Alarm_H_
3  #include "framework/utils/SingleInclude.h"
4  class Alarm : public FBE {
5    public:
6      Alarm();
7      ~Alarm();
8      Boolean * atOn;
9      Integer * atTimer;
10     Method * mtRingTheBell;
11 };
```

Fonte: Autoria Própria

Conforme apresenta o Código 24, o gerador de código cria uma classe em linguagem C++ que segue o modelo de declarações do *Framework* PON C++ 2.0. Para esse código em especial foi gerado um conjunto de declarações do *FBE Alarm*, contendo seus respectivos *Attributes* e *Methods.*

O Código 25, por sua vez, apresenta o código do arquivo Alarm.cpp, contendo as definições da classe em questão, instanciando e inicializando as respectivas entidades PON, com base nas definições disponíveis no *Framework* PON C++ 2.0[15].

Código 25: Código gerado para as definições do arquivo Alarm.cpp

```
1  #include "Alarm.h"
2  Alarm::Alarm(void) {
3    BOOLEAN(this,atOn,false);
4    INTEGER(this,atTimer,0);
5    METHOD(this,mtRingTheBell,atTimer,true,60);
6  }
7  Alarm::~Alarm(void) {
8  }
```

Fonte: Autoria Própria

Nesse exemplo, em particular, os códigos em questão apresentam apenas a criação dos arquivos para o *FBE Alarm*. Entretanto, para cada *FBE* definido no programa, é criado um conjunto de respectivos arquivos de declarações e definições, os quais representam as particularidades de cada qual. Além disso, também é gerado

[15] Detalhes de implementação do *Framework* PON C++ 2.0, bem como um conjunto de boas práticas para a criação de programas seguindo a estrutura da ferramenta, podem ser encontrados em (RONSZCKA, 2012).

um conjunto de arquivos para representar o código de inicialização do programa, bem como o conjunto de *Rules* definido para a execução do mesmo. Nesse âmbito, o Código 26 apresenta um trecho de código gerado com as declarações de inicialização de um programa em PON para o *Framework* PON C++ 2.0.

Código 26: Trecho de código com as declarações do arquivo Main.h

```
1  class Main : public NOPApplication {
2
3    public:
4      Main();
5      virtual ~Main();
6
7    public:
8      void initStartApplicationComponents();
9      void initFactBase();
10     void initRules();
11     void initSharedEntities();
12     void codeApplication();
13
14   public:
15     RuleObject * rlFireAlarm;
16     Premise* prSensorPresence1;
17     Premise* prSensorPresence2;
18     Instigation* inFireAlarm;
19     Alarm * alarm1
20     SensorPresence * sensorPresence1;
21     SensorPresence * sensorPresence2;
22 };
```

Fonte: Autoria Própria

Conforme apresenta o Código 26, mais especificamente nas linhas 15 a 21, todos os elementos necessários são declarados na classe *Main*, tais como as *Rules*, *Premises*, *Instigations* e *FBEs* pertinentes a execução do programa. Ainda, no arquivo Main.cpp, no qual se encontram as definições de cada elemento declarado, são feitas as conexões de notificação entre as entidades em questão.

De modo a continuar a exemplificar a montagem de um programa para *Framework* PON C++ 2.0, o Código 27 apresenta agora as particularidades da criação de uma *Rule*, mais precisamente, a *Rule rlFireAlarm*.

Código 27: Trecho de código gerado para a definição de *Rules*

```
1  void Main::initRules() {
2    Scheduler * scheduler = SingletonScheduler::getInstance();
3    RULE (rlFireAlarm,scheduler,Condition::CONJUNCTION);
4          rlFireAlarm->addPremise(prSensorPresence1);
5          rlFireAlarm->addPremise(prSensorPresence2);
6          rlFireAlarm->addInstigation(inFireAlarm);
7          rlFireAlarm->end();
8  }
```

Fonte: Autoria Própria

Conforme apresenta o Código 27, o método *initRules* inicialmente recupera a instância única do escalonador (*scheduler*), escolhido na definição da estratégia do programa em LingPON. Na sequência, são definidas as *Rules* do programa com base nesse escalonador. No exemplo em questão, a *Rule rlFireAlarm* apresenta uma conexão de conjunção (*and*) com as duas *Premises* pertinentes (*i.e., prSensorPresence1* e *prSensorPresence2*) que permitem internamente criar a *Condition* dela. A *Rule* em questão ainda é composta pela *Instigation inFireAlarm*, a qual representa a parte execucional da *Rule*, permitindo internamente criar a *Action* dela.

De maneira geral, além de testar o sistema de compilação como um todo, o objetivo dessa versão de geração de código foi automatizar a criação de programas aptos a executar no *Framework* PON C++ 2.0, simplificando essa etapa, uma vez que a programação para o *framework* demanda um certo grau de conhecimento de sua estrutura interna e funcionamento. Ademais, é importante salientar que toda a geração de código apresentada segue os padrões e diretrizes de boas práticas definidos em (RONSZCKA, 2012)[16].

5.1.2.3.2 Gerador de código para linguagem C - Versão C *específica a notificações*

O gerador de código para linguagem C apresenta uma estrutura particular, estritamente procedimental, na qual cada relação notificante entre dois elementos PON é representada por uma função. No corpo de cada função estão todos os elementos interessados na mudança de estado do elemento notificante, o qual realiza chamadas para outras funções conforme o fluxo de execução converge. Na prática, o gerador de código dessa versão cria uma corrente de chamadas de funções que representam o fluxo de notificações entre todos os elementos do programa (RONSZCKA *et al*., 2013; RONSZCKA *et al*., 2017b).

[16] Em (RONSZCKA, 2012) são apresentados alguns padrões de implementação, bem como um conjunto de diretrizes e boas práticas para a programação em PON. Em linhas gerais, os padrões de implementação representam a utilização de nomenclatura adequada para com os elementos de software, correto uso de comentários, formatação e organização do código como um todo. Nesse âmbito, uma das recomendações, em especial, é o uso de prefixos padronizados nos nomes das entidades para melhorar a legibilidade do código, tais como: o prefixo at para nomenclatura de *Attributes*, pr para *Premises*, cd para *Conditions*, rl para *Rules*, ac para *Actions*, in para *Instigations* e mt para *Methods*.

Nesse contexto, a geração de código para C foi pensada e desenvolvida de maneira a simplificar a estrutura do programa gerado, não fazendo uso de passagem de parâmetros, alocação dinâmica de memória e estrutura de dados. Dessa forma, a compilação do código gerado em C para código de máquina, a cargo do compilador GCC, tenderia a ser mais eficiente[17] (FERREIRA, 2015).

Em linhas gerais, a geração de código em C define estruturas de dados do tipo *struct* para armazenar o estado de aprovação de três entidades do PON (*i.e., Premise*, *Subcondition* e *Rule*). Com base nessa estrutura, são criadas instâncias para cada elemento presente no código compilado. O Código 28, apresenta essa estrutura.

Código 28: Trecho de código gerado para a definição de *Rules* na compilação para C específico a notificações

```
1  typedef struct RuleType {
2     short isApproved;
3  } Rule;
4  Rule RlFireAlarm, RlTurnOn, RlTurnOff;
```

Fonte: Autoria Própria

Conforme apresenta o Código 28, a estrutura *RuleType* define uma variável do tipo *short* para representar o status de aprovação (*isApproved*) de uma *Rule*. Ainda, no exemplo, mais precisamente na linha 4, são criadas três instâncias de *Rules* pertinentes ao programa Sensores. Essa estrutura possui escopo global e é utilizada como base fundamental para o fluxo de execução do programa gerado. Outrossim, as estruturas *Premise* e *Subcondition* são parecidas a da *Rule*, possuindo suas próprias estruturas, *PremiseType* e *SubconditionType*, respectivamente.

A geração de código nessa versão cria arquivos independentes com o objetivo de facilitar o entendimento de cada uma das partes individualmente. Além disso, para cada elemento do programa é criado um arquivo de cabeçalho (.h) que contém as declarações das estruturas e cabeçalhos de funções, bem como um arquivo de definições (.c) que implementa os procedimentos e as funções pertinentes àquela entidade em questão. Nesse âmbito, o Código 29 apresenta um exemplo de geração

[17] Na verdade, apesar das otimizações realizadas pelo compilador GCC, nem todos os "defeitos" de um código mal projetado são consertados em tempo de compilação. Nesse sentido, utilizar boas práticas para a composição de programas bem estruturados, que evitem a utilização de técnicas e estruturas de dados dispensiosas computacionalmente, normalmente causam um impacto positivo em termos de desempenho na execução de tais programas.

de código para uma instância de um *SensorPresence* e toda a sua cadeia de notificações baseada em funções.

Código 29: Cadeia de notificações gerada em linguagem C

```
1  void setSensorPresence1atStatus(bool pValue){
2    if (pValue != sensorPresence1.atStatus) {
3      sensorPresence1.atStatus = pValue;
4      funcInstsensorPresence1AtatStatusNotifyPrprSensorPresence1() {
5    }
6  }
7  void funcInstsensorPresence1AtatStatusNotifyPrprSensorPresence1() {
8    if (sensorPresence1.atStatus == true) {
9      setPrprsensorPresence1isApproved(1);
10     funcInstsensorPresence1PrprSensorPresence1NotifyScA2();
11   } else {
12     setPrprsensorPresence1isApproved(0);
13   }
14 }
15 void funcInstsensorPresence1PrprSensorPresence1NotifyScA2() {
16   if (prSensorPresence1.isApproved & prSensorPresence2.isApproved){
17     setScA2isApproved(1);
18     funcInstsensorPresence1ScA2NotifyRlRlFireAlarm();
19   } else {
20     setScA2isApproved(0);
21   }
22 }
23 void funcInstsensorPresence1ScA2NotifyRlRlFireAlarm(){
24   if (A1.isApproved & A2.isApproved) {
25     setRlRlFireAlarmisApproved(1);
26     instsensorPresence1mtRingTheBell();
27   } else {
28     setRlRlFireAlarmisApproved(0);
29   }
30 }
31 void instsensorPresence1mtRingTheBell(){
32   setsensorPresence1atTimer(60);
33 }
```

Fonte: Autoria Própria

Conforme apresenta o Código 29, o fluxo de execução do programa se dá da seguinte forma: A cada mudança de um *Attribute* (linhas 1 a 6), todas as *Premises* interessadas são avaliadas. No caso de serem aprovadas, configuram a variável (*isApproved*) de sua instância como verdadeira (linha 9). Em seguida, a *SubCondition* que contém aquela *Premise* é notificada. Uma avaliação bit a bit das variáveis que indicam a aprovação das *Premises* existentes naquela *SubCondition* é realizada para verificar se ela for aprovada (linha 16). No caso dela ser aprovada, a *Rule* interessada é notificada. Em seguida, a função que avalia a *Rule* verifica se todas as *SubConditions* existentes nela estão válidas (linha 24). Se sim, a *Action* indicadas pela *Rule* é executada finalizando assim uma etapa (linha 26). Após essa execução, pode

ocorrer outra mudança de *Attribute* que faz com que a cadeia seja disparada novamente.

De maneira geral, essa versão de geração de código apresenta uma estrutura bastante simplificada para definir a cadeia de notificações em forma de funções, com o auxílio de algumas estruturas simplificadas para armazenar o estado atual de entidades. Nesse modelo de código gerado foi possível eliminar a presença de classes e estruturas de dados (como listas e vetores), criando um código totalmente estático, sem alocação dinâmica de memória. Entretanto, é importante salientar que essa versão apresenta alguns problemas de legibilidade para com o código gerado, dificultando o entendimento do mesmo se alguém assim o desejasse. Além disso, o código apresenta um problema de redundância estrutural, uma vez que a cada chamada da função de aprovação de uma *SubCondition*, a verificação de aprovação de uma *Rule* é sempre reavaliada (linha 24).

5.1.2.3.3 Gerador de código para C++ - Versão C++ *específica a notificações*

O gerador de código específico orientado a notificações em linguagem C++, por sua vez, apresenta uma estrutura particular que se utiliza de alguns recursos da orientação a objetos. Nesse âmbito, as classes seguem um padrão de simplificação da estrutura tradicional aplicada no *Framework* PON C++ 2.0, resumindo os elementos fundamentais da estrutura do paradigma em três classes principais, que são: *FBE*, *Premise* e *Rule*.

Entretanto, diferentemente do *Framework* PON C++ 2.0, o qual possuía uma classe genérica para cada uma das entidades PON, o gerador de código em C++ cria classes específicas para cada uma das três entidades (*FBE*, *Premise* e *Rule*). Assim, tais classes seguem um modelo estritamente pontual, definindo as relações notificantes de forma concreta, ao invés de utilizar listas dinâmicas para manter tais relações, sendo esta última a utilizada pelo *Framework* PON C++ 2.0, conforme apresentado no Capítulo 3.

De modo a apresentar as principais classes desse modelo e seus relacionamentos por notificações, os códigos 30 a 32 mostram trechos de código de entidades criadas a partir da compilação do programa Sensores.

Código 30: Trecho de código com a estrutura de um FBE compilação para C++ específico a notificações

```
1  class Alarm {
2    public:
3      Alarm(void);
4      bool atOn;
5      void setatOn(bool atOn);
6      int atTimer;
7      void setatTimer(int atTimer);
8      void mtRingTheBell();
9      PrAlarmOn prAlarmOn;
10 };
11
12 Alarm::Alarm(void) {
13   atOn = false;
14   atTimer = 0;
15 }
16 void Alarm::setatOn(bool atOn) {
17   if (this->atOn != atOn) {
18     atOn = atOn;
19     prAlarmOn.setatOn(atOn);
20   }
21 }
22 void Alarm::setatTimer(int atTimer) {
23   if (atTimer != atTimer) {
24     atTimer = atTimer;
25   }
26 }
27 void Alarm::mtRingTheBell() {
28   setatTimer(60);
29 }
```

Fonte: Autoria Própria

Conforme apresenta o Código 30, o trecho de código mostra a estrutura de um *FBE* gerado para o modelo de compilação C++ *específica a notificações*. Nesse modelo, as linhas 1 a 10 apresentam a estrutura da classe e suas declarações, enquanto as linhas 12 a 29 apresentam as definições da classe.

Em linhas gerais, essa classe é composta por seus respectivos *Attributes* e *Methods*, assim como uma instância concreta de uma *Premise* (linha 9) que faz relação com os *Attributes* do *FBE*. Nesse caso, quando ocorre a mudança de um *Attribute*, esse notifica prontamente as *Premises* interessadas via chamada direta, conforme mostra a linha 19. Na linha 28, o *Method* do *FBE* faz a alteração do estado de um *Attribute* para um valor predeterminado, conforme definido no programa escrito em LingPON.

É pertinente ressaltar novamente, mas agora à luz do exemplo, que nesse modelo de geração de código, a relação entre *Attributes* e *Premises* difere da implementação baseada em listas dinâmicas do *Framework* PON C++ 2.0. No modelo em questão, cada entidade do tipo *Attribute* conhece exatamente as *Premises* que se

interessam pelo seu estado e a conexão entre elas é feita de forma concreta, sem o uso de ponteiros em estruturas de dados (*e.g. Attribute* com lista encadeada de ponteiros de *Premises*). Sendo assim, o código final gerado se adequa as particularidades de cada programa em tempo de montagem e compilação. Ainda, para exemplificar a estrutura de uma *Premise*, o trecho de código do Código 31 apresenta as particularidades dessa entidade.

Código 31: Trecho de código com a estrutura de uma *Premise* compilação para C++ específico a notificações

```
1  class PrAlarmOn {
2    public:
3      PrAlarmOn(void);
4      bool isApproved;
5      void setatAtOn(bool atOn);
6      RlTurnOn * rlTurnOn;
7  };
8
9  void PrAlarmOn::setatAtOn(bool atOn) {
10   if (atOn == false) {
11     isApproved = true;
12     rlTurnOn->isApproved();
13   } else {
14     isApproved = false;
15   }
16 }
```

Fonte: Autoria Própria

Conforme apresenta o Código 31, uma *Premise* possui um *Attribute* que armazena seu estado lógico atual (linha 4) e um método para a validação de seu respectivo cálculo lógico, na qual a declaração está na linha 5 e a definição nas linhas 9 a 16. Ainda, a linha 6 apresenta a referência para uma *Rule*, a qual possui interesse na mudança de estado lógico da *Premise* ilustrada nesse exemplo em particular. Caso a verificação do estado lógico da *Premise* seja satisfeito, a verificação de aprovação da *Rule* é chamada (linha 12). Nesse âmbito, o Código 32 apresenta o trecho de código referente a estrutura de uma *Rule*.

Código 32: Trecho de código com a estrutura de uma *Rule* compilação C++ específico a notificações

```
1  class R1TurnOn {
2    public:
3      R1TurnOn(void);
4      bool isApproved();
5      SensorPresence * sensorPresence1;
6      SensorPresence * sensorPresence2;
7  };
8
9  bool R1TurnOn::isApproved() {
10   if (sensorPresence1->isApproved & sensorPresence2->isApproved) {
11     sensorPresence1->mtTurnOn();
12     sensorPresence2->mtTurnOn();
13   }
14 }
```

Fonte: Autoria Própria

Conforme apresenta o Código 32, a *Rule* em questão possui duas *Premises*, as quais precisam estar igualmente com seu estado lógico verdadeiro para que a aprovação da *Rule* seja efetiva. Uma vez aprovada, os *Methods* definidos são executados, conforme apresenta as linhas 11 e 12.

O gerador de código orientado a objetos em linguagem C++ tornou possível apresentar algumas otimizações ao evitar o uso de estruturas de dados e objetos dinâmicos em sua essência. Em contrapartida, o código apresentou uma certa imaturidade, principalmente no fato de fazer verificações causais redundantes, como no caso do Código 32, mais especificamente na linha 10, na qual cada chamada ao método *isApproved* precisa reavaliar suas próprias *Premises* antes de ter a execução da *Rule* efetivada.

5.1.3 Testes de desempenho e de uso de memória nos programas compilados

Um dos motivadores da criação de Tecnologia LingPON (prototipal e subsequentes) seria buscar alcançar as propriedades elementares do PON tanto quanto a plataforma computacional visada permitisse. Neste sentido, nessa Tecnologia LingPON Prototipal, observou-se que foi possível finalmente criar código PON em alto nível, efetivamente orientado a regras. Ainda, para o caso dado, também se fez pertinente experimentos de performance.

Conforme apresentado em (RONSZCKA *et al*., 2013) foram realizados testes preliminares de desempenho e de uso de memória nos três alvos gerados pela LingPON para um mesmo programa, lembrando que naquela feita o *Framework PON*

C++ 2.0 (um dos alvos dos geradores de código criados) era a materialização mais performante. Em tempo, certamente que o entendimento dos experimentos que seguem não é fundamental para a compreensão da aplicação do MCPON Preliminar já relatada, sendo eles aqui apresentados como conteúdo suplementar, à luz das motivações da Tecnologia LingPON.

Isto dito, o programa utilizado nos experimentos é chamado de Mira ao Alvo. Este programa constitui-se em um *toy problem* no qual arqueiros devem atirar flechas em maçãs dada uma condição de prontidão, sendo que sua vantagem fundamental é permitir testes em escala (BANASEWSKI, 2009; SIMÃO *et al.*, 2012a). Na verdade, se for analisado apropriadamente, esse *toy problem* seria semanticamente equivalente a programas que correlacionam estados de entidades, como um programa que correlaciona estados de alarmes, como o programa exemplo usado em (RONSZCKA *et al.,* 2015).

5.1.3.1 Caso de Estudo - Programa Mira ao Alvo

O programa Mira ao Alvo explora, principalmente, o fato de existirem redundâncias estruturais (comparar quais arqueiros podem atirar e quais não podem) e redundâncias temporais (comparações repetidas e desnecessárias até que os arqueiros estejam prontos) em programas baseados em linguagens e paradigmas usuais. Ademais, esse programa tem sido utilizado como ambiente para testes de *benchmark* nas ferramentas e técnicas desenvolvidas pelo grupo de pesquisa do PON, uma vez que abstrai bem a essência de programas de maior escala, nos quais as redundâncias tendem a estar dispersas na estrutura do programa (RONSZCKA *et al*., 2017a).

Basicamente, a aplicação consiste em um ambiente no qual as entidades do tipo ‘Mira’ interagem ativamente com as entidades do tipo ‘Alvo’. Mais precisamente, as entidades ‘Mira’ e as entidades ‘Alvo’ são representadas, respectivamente, por arqueiros e maçãs, sendo que há uma maçã para cada arqueiro, no cenário dado.

Em termos de implementação, cada arqueiro e maçã são representados por meio de instâncias de *FBEs*, os quais interagem de acordo com a validação de suas expressões causais pertinentes na forma de *Rules*. Estas *Rules* correlacionam os estados de estados de Atributos destas instâncias de FBE, o que permite ativar seus *Methods* caso a correção gera aprovação da *Rule*.

Nesse contexto, cada *FBE Arqueiro* e cada *FBE Maçã* recebem um identificador numérico, sendo que um *FBE Arqueiro* somente pode flechar um *FBE Maçã* que apresente o identificador numérico correspondente ao seu. Ainda, cada *FBE Arqueiro* também apresenta um *Attribute* que denota o estado de pronto (*i.e.,* status) para agir sobre o cenário (*i.e.,* flechar a respectiva maçã). Outrossim, um terceiro elemento, denominado *FBE Arma de Fogo*, tem a função de sinalizar com o seu disparo o início de uma iteração, permitindo que os arqueiros interajam com as respectivas *FBE Maçãs*.

Em relação as *FBE Maçã*, estas também apresentam cada qual um *Attribute* que denota o seu estado de pronto (*i.e., status*). Ainda, essas também apresentam um *Attribute* que explicita se a mesma já foi perfurada por uma flecha ou não (*i.e., isCrossed*). Também, cada *FBE Maçã* apresenta um *Attribute* que se refere a sua coloração (*i.e., color*), uma vez que cada *FBE Maçã* pode se apresentar em duas diferentes cores: vermelha ou verde.

Ademais, cada FBE Arqueiro somente pode interagir com a sua respectiva *FBE Maçã* após a constatação de quatro premissas: (a) se a cor do respectivo *FBE Maçã* que está posicionada diretamente em sua frente é vermelha, (b) se o respectivo *FBE Maçã* que está posicionada diretamente em sua frente está pronta para ser atingida e (c) o estado do FBE Arma de Fogo é atirar.

Em termos de PON, cada grupo de quatro *Premises* em PON, nestes moldes de avaliação de instâncias, permitem compor a *Condition* de cada *Rule* no programa em questão. Para o caso em que as quatro *Premises* estiverem satisfeitas, o FBE Arqueiro está liberado para atingir o respectivo FBE Maçã com a pretensa projeção de sua flecha. Dessa forma, para cada par de FBE Arqueiro e FBE Maçã deve haver uma expressão causal (*i.e., Rule*) para comparar os seus estados.

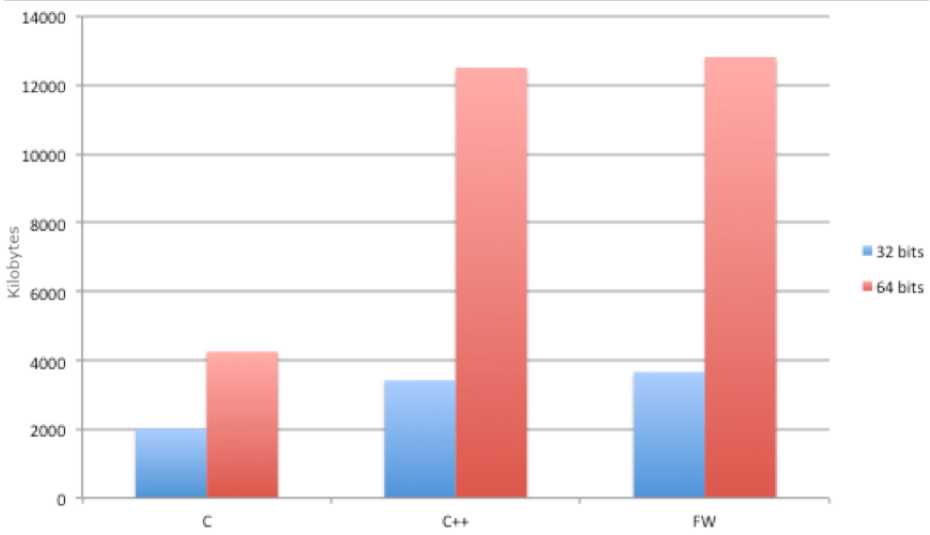
5.1.3.2 Experimentos sobre o programa Mira ao Alvo

Com base no programa Mira ao Alvo, de modo a realizar comparações de eficiência entre as três versões de código geradas, um estudo comparativo de desempenho foi realizado para três cenários: 1, 10 e 100 *Rules*, sendo que para cada um destes cenários foram executadas 1, 10, 100, 1.000, 10.000 e 100.000 iterações. Os testes foram realizados em duas arquiteturas de sistemas operacionais (*i.e.,* 32 e 64 bits) (RONSZCKA *et al*., 2013).

Os tempos foram considerados apenas ao final da execução de cada programa (*i.e.,* após todas as iterações). Para realizar a aferição dos testes foi utilizado um ambiente Linux Debian 7.1 (32 e 64 bits) em uma máquina com 4 GB RAM, Intel Core 2 Quad CPU Q6600 @ 2.40 GHz x 4. Todos os testes foram realizados em um ambiente livre de preempção (Linux monousuário/modo de recuperação), os códigos foram compilados com GCC (linguagem C) e G++ (linguagem C++ e *Framework PON C++ 2.0*) (RONSZCKA *et al*., 2013).

No âmbito de testes de uso de memória, os valores apresentados no gráfico da Figura 59 apresentam uma síntese dos resultados no cenário com 100 *Rules*. É importante salientar que os valores são apresentados em Kilobytes.

Figura 59: Comparação de uso de memória no cenário com 100 *Rules*
Tecnologia LingPON Prototipal: "C" (C específica a notificações), "C++" (C++ específica a notificações), "FW" (Framework PON C++ 2.0)

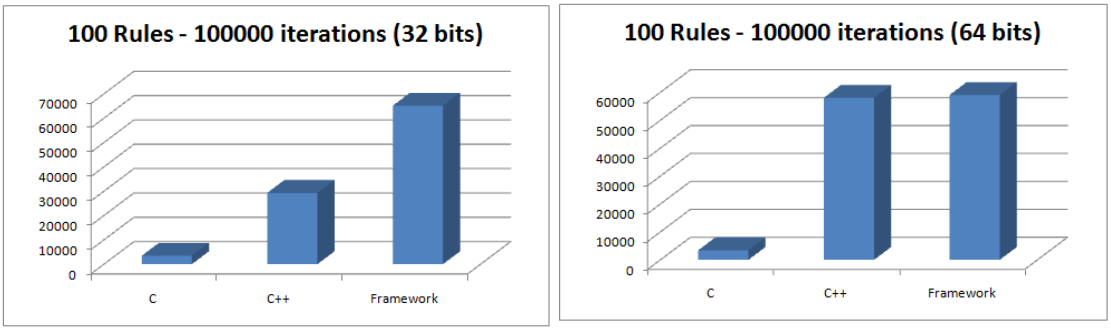


Fonte: Valores extraídos de Ronszcka et al., 2013

Conforme apresenta a Figura 59, é possível observar que a execução do conjunto de testes no sistema em 64 bits utilizou, em média, aproximadamente quatro vezes mais memória que os executados em 32 bits. Também foi constatado que para cada cenário de testes (com 1, 10 ou 100 *Rules*) os resultados aferidos não alteravam com relação ao número de iterações, ou seja, sempre eram constantes os usos de memória independentemente do número de iterações executadas (RONSZCKA *et al*., 2013). Além disso, é importante notar que a versão em C *específica a notificações* apresentou uma melhor utilização de memória em comparação as outras versões. Isso se deu principalmente pelo fato de que na versão C *específica a notificações* não é utilizada alocação dinâmica de memória para a criação dos elementos notificantes.

Em relação aos testes de desempenho de execução, a Figura 60 apresenta apenas os resultados do experimento para 100 *Rules* e 1 milhão de iterações, em ambos os ambientes (32 e 64 bits). É importante ressaltar que os resultados completos dos experimentos são encontrados em (RONSZCKA, *et al*., 2013).

Figura 60: Comparação de tempo de execução no cenário com 100 *Rules* Tecnologia LingPON Prototipal: "C" (C específica a notificações), "C++" (C++ específica a notificações), "Framework" (Framework PON C++ 2.0)



Fonte: Adaptado de Ronszcka et al., 2013

Para o conjunto de testes em 32 bits, os resultados da versão em C *específica a notificações* em comparação com os resultados do *Framework* PON C++ 2.0 foram, em média, aproximadamente 13 vezes mais eficientes em termos de tempo de execução. Em relação ao *Framework* PON C++ 2.0 comparado com a versão C++ *específica a notificações*, os resultados deste foram, em média, aproximadamente duas vezes mais eficientes. Para o conjunto de testes em 64 bits a versão C++ *específica a notificações* apresentou tempos próximos aos do *Framework PON C++ 2.0*. Em relação a versão C *específica a notificações*, os resultados se mantiveram, apresentando a mesma eficiência média que no ambiente 32 bits. Em suma, a versão em C *específica a notificações* conseguiu executar com muito menos tempo que C++ *específica a notificações* e *Framework* PON C++ 2.0 em praticamente todos os cenários de testes, conforme apresentado em (RONSZCKA *et al*., 2013).

5.1.4 Considerações sobre a Tecnologia LingPON Prototipal

Em linhas gerais, o ferramental que compõem a tecnologia LingPON (i.e., Sistema de Compilação Preliminar, a LingPON prototipal em si e os três geradores de código explicados), foram amparados pelo método MCPON em uma versão preliminar. Neste sentido, para a concepção da linguagem foram seguidas as diretrizes vislumbradas nas etapas 1 e 2 do método MCPON.

As etapas 1 e 2 do MCPON basicamente norteiam a construção de grafos especializados com base em um conjunto de entidades instanciáveis à luz do Grafo PON. Conforme determina o método, a LingPON deve de fato ter a fase de análises integrada a composição do grafo, conforme detalha a Seção 5.1.2.2. Posteriormente, foram construídos três geradores de código baseados nos grafos extraídos da fase de análise. É importante observar que mesmo nesta versão prototipal do ferramental apresentado, já foi possível separar a fase de análises da fase de síntese, por meio da representação intermediária Grafo PON prototipal.

Cada uma das três versões de geradores de código criados foi submetida a testes de desempenho e de memória. Em termos de eficiência de execução, a versão de código gerado para C *específica a notificações*, cujo fluxo de notificações é orientado a funções, apresentou melhores resultados em relação aos outros *targets* apresentados previamente. Basicamente, nessa versão foram adaptados os conceitos do PON para uma implementação menos onerosa (eliminação de classes e estruturas de dados) baseando a implementação no paradigma estruturado (RONSZCKA *et al*., 2013).

De maneira geral, os resultados em relação a criação de um sistema de compilação para o PON se mostraram bastante promissores. Os esforços para a criação de uma linguagem própria para o PON e a implementação de um sistema de compilação à luz do método MCPON, com distintas versões de geradores de código final, confirmam o potencial de compilação para que o PON atinja suas reais capacidades principalmente em termos de desempenho (RONSZCKA *et al*., 2013).

Apesar desses esforços iniciais, a LingPON prototipal e o Sistema de Compilação Preliminar, possibilitaram apenas a criação de programas básicos (*i.e.,* execução de *Methods* com atribuições simples). Nesse âmbito, esforços em prol de adicionar outras funcionalidades essenciais eram desejáveis (*e.g. Methods* com operações aritméticas e *Methods* para integração com códigos legados em C/C++),

de modo a possibilitar a criação de programas mais elaborados. Para isso, foi desenvolvida a versão chamada de 1.0, conforme apresenta a próxima subseção.

5.2 Tecnologia LINGPON 1.0 – evoluções da linguagem e geradores de código

De maneira geral, o trabalho de mestrado de Ferreira (2015) foi reorganizar todo o ferramental gerado na versão prototipal da linguagem e do sistema de compilação de modo a documentar o que havia sido criado prototipalmente a partir dos esforços iniciais do grupo de pesquisa do PON. Além disso, Ferreira implementou alguns dos conceitos do PON faltantes na versão prototipal.

Para a implementação de tais conceitos foi necessário adicionar novas palavras reservadas a LingPON e adequar os analisadores léxico e sintático para tratar os novos símbolos adequadamente. Além disso, foi necessário adicionar novos elementos no Grafo PON de modo a mapear fidedignamente tais conceitos.

Com base nesses avanços foi definida a versão 1.0 da linguagem LingPON e dos respectivos geradores de código para *Framework* PON C++ 2.0 e para código C *específico a notificações* e C++ *específico a notificações*, constituindo a Tecnologia LingPON 1.0, mas sem modificação elementar no Sistema de Compilação Preliminar em si. De modo a apresentar as evoluções do trabalho de Ferreira, as subseções seguintes apresentam detalhes sobre as melhorias propostas.

5.2.1 Evoluções na linguagem LingPON e dos geradores de código associados

No âmbito das evoluções na linguagem LingPON e dos respectivos geradores de código, à luz do Sistema de Compilação Preliminar e com base na Tecnologia LingPON Prototipal, o trabalho de Ferreira (2015) contribuiu com a implementação de alguns dos conceitos que não haviam sido ainda implementados na versão preliminar, tais como a inclusão de propriedades das *Rules*, impertinência de entidades e dois novos blocos principais: (1) um para definição da estratégia de escalonamento de *Rules*; e (2) outro para a definição de atribuições iniciais (FERREIRA, 2015). Para isso, Ferreira precisou se amparar nas mesmas etapas e subetapas do método MCPON Preliminar utilizados na Tecnologia LingPON Prototipal. Tais modificações se consolidaram na chamada Tecnologia LingPON 1.0 e são apresentadas na *BNF* completa exposta no apêndice C.

5.2.1.1 Implementação do conceito de Propriedades das *Rules*

Em relação aos conceitos adicionais implementados na versão 1.0, as *Rules* passaram a ter um sub-bloco de propriedades, permitindo ao desenvolvedor definir algumas características adicionais para tais entidades. Esse bloco é opcional e pode ser utilizado em um conjunto de *Rules* específico. De modo a demonstrar a utilização desse bloco, o Código 33 apresenta um exemplo de utilização em uma *Rule*.

Código 33: Exemplo da utilização do bloco opcional *properties*

```
1  rule rlFireAlarm
2    properties
3      priority 1
4      keeper true
5    end_properties
6    conditions
7    end_conditions
8      . . .
9    action
10     . . .
11   end_action
12 end_rule
```

Fonte: Autoria Própria

Conforme apresenta o Código 33, o exemplo apresenta um exemplo de *Rule* destacando o bloco de propriedades. Nesse bloco opcional, anunciado pelas palavras reservadas *properties* e *end_properties*, o desenvolvedor pode informar a prioridade de execução de determinada *Rule* e se a *Rule* apresenta a funcionalidade *Keeper*.

A prioridade é definida por meio da palavra reservada *priority*, seguido de seu respectivo valor de indicação de prioridade, conforme apresenta a linha 3. A propriedade *priority* define a ordem de prioridade de execução quando duas ou mais *Rules* compartilham do mesmo *Exclusive Attribute* em alguma de suas *Premises*. Além disso, quando a estratégia de escalonamento de *Rules* é definida como sendo do tipo PRIORITY, o escalonador de *Rules* utiliza do valor definido em cada qual para organizar a execução do programa. Sendo assim, a *Rule* que apresentar a maior prioridade vai ser priorizada em relação as demais.

A propriedade *keeper*, por sua vez, define o comportamento reativo de execução da *Rule* particular, permitindo postergar a execução da ação de uma *Rule* para outro momento. Em tempo, certamente que a cada novo conceito introduzido na linguagem LingPON, como este conceito de *Keeper*, os respectivos geradores de código tinham que ser igualmente atualizados à luz do MCPON preliminar.

5.2.1.2 Implementação do conceito de Entidades Impertinentes

Ainda, outro conceito implementando na versão 1.0 que apresenta um impacto considerável em termos de execução foi o de Entidades Impertinentes (conforme conceito explicado na Seção 3.5.3). Para definir a impertinência de um dado *Attribute* em uma determinada *Premise*, é necessário utilizar pontualmente a palavra reservada *imp* apenas na *Premise* desejada. O Código 34 apresenta o uso correto dessa funcionalidade.

Código 34: Exemplo da utilização do bloco opcional *properties*

```
1 premise prSensorTemp1 imp sensorTemp1.temperature > 30
```

Fonte: Autoria Própria

Para cada código-alvo gerado é criado um mecanismo interno, específico para cada implementação que normalmente bloqueia a propagação das notificações do *Attribute* impertinente para a *Premise* definida com a propriedade *imp*. Nesse caso, ela fica condicionada a uma pré-aprovação da *Rule* a qual faz parte, na qual todas as demais *Premises* da *Rule* devem estar aprovadas, para só então verificar o estado atual da *Premise* marcada com a impertinência.

5.2.1.3 Novo bloco principal *strategy*

Um conceito importante para a criação de programas em PON é a definição da estratégia de escalonamento de *Rules* a ser utilizada em tempo de execução, quando duas ou mais *Rules* se tornarem aprovadas ao mesmo tempo. Para exemplificar a definição da estratégia a ser utilizada, o Código 35 exemplifica o uso particular da estratégia BREADTH.

Código 35: Exemplo de definição de Estratégia de Escalonamento da LingPON

```
1  fbe
2  end_fbe
3
4  inst
5  end_inst
6
7  strategy
8    breadth
9  end_strategy
10
11 rule
12 end_rule
```

Fonte: Autoria Própria

Conforme apresenta o Código 35, a definição consiste em apenas informar o nome da estratégia a ser utilizada na compilação do programa. Em suma, a versão preliminar da linguagem possui quatro estratégias de escalonamento disponíveis, que são: (a) *no_one* – na qual nenhuma estratégia é aplicada, sendo assim, as *Rules* executam a medida em que são aprovadas; (b) *breadth* – estratégia de execução em *FIFO* (*First In-First Out*), na qual as *Rules* são ordenadas em uma fila; (c) *depth* – estratégia de execução em *LIFO* (*Last In-First Out*), na qual as *Rules* são ordenadas em uma pilha; e (d) *priority* – na qual as *Rules* são ordenadas baseado em um indicador de prioridade definido em cada *Rule* (conforme Seção 3.5.1).

É importante ressaltar que, na versão 1.0 da LingPON, o desenvolvedor deve definir obrigatoriamente uma estratégia de escalonamento de *Rules* para seus programas. Por se tratar de um programa compilado, a definição da estratégia é um elemento importante, uma vez que a ordem das notificações ficará enraizada na estrutura do código final gerado, garantindo uma execução determinística em programas compilados para um único fluxo de execução, isto é, processados unicamente pela *thread* principal. Ademais, a abertura e fechamento da seção é representada pelas palavras reservadas ***strategy*** e ***end_strategy***.

5.2.1.4 Novo bloco principal *main*

Por fim, Ferreira (2015) definiu um novo bloco na estrutura geral de um programa, o bloco *main*, no qual é possível adicionar código específico da linguagem alvo escolhida no processo de compilação (*e.g.* C ou C++). Um exemplo de utilização do bloco é apresentado no Código 36.

Código 36: Exemplo da utilização do bloco *main*

```
1  fbe
2  end_fbe
3
4  inst
5  end_inst
6
7  strategy
8  end_strategy
9
10 rule
11 end_rule
12
13 main {
14   apple->setatOn(false);
15 }
```

Fonte: Autoria Própria

Na prática, o bloco *main* é utilizado para a inicialização do programa e definição das primeiras atribuições de valores aos *Attributes*. Com isso, é possível dar início ao fluxo de execução baseado nas notificações subsequentes ao valor atribuído. É importante salientar que o conteúdo do bloco *main* não é interpretado e traduzido para a linguagem-alvo, sendo apenas transcrito para o programa compilado. Nesse caso, é preciso programar o código conforme a linguagem alvo. No exemplo em questão, conforme apresenta a linha 14, o código em questão está voltado para a versão C++ *específica (orientada) a notificações*.

5.2.2 Experimentos na LingPON 1.0

Em linhas gerais, o trabalho de Ferreira (2015) apresenta experimentos comparativos entre as três versões de código gerado, avaliando características como desempenho de execução, quantidade de linhas do código gerado em *Assembly* e o uso de memória. Adicionalmente, analisou-se a facilidade de programação de aplicações para o PON entre a LingPON e o *Framework* PON C++ 2.0, por meio da análise em termos de linhas de códigos e número de *tokens*. Além disso, o trabalho apresenta relatos de desenvolvedores que utilizaram as duas abordagens para o desenvolvimento de aplicações diversas.

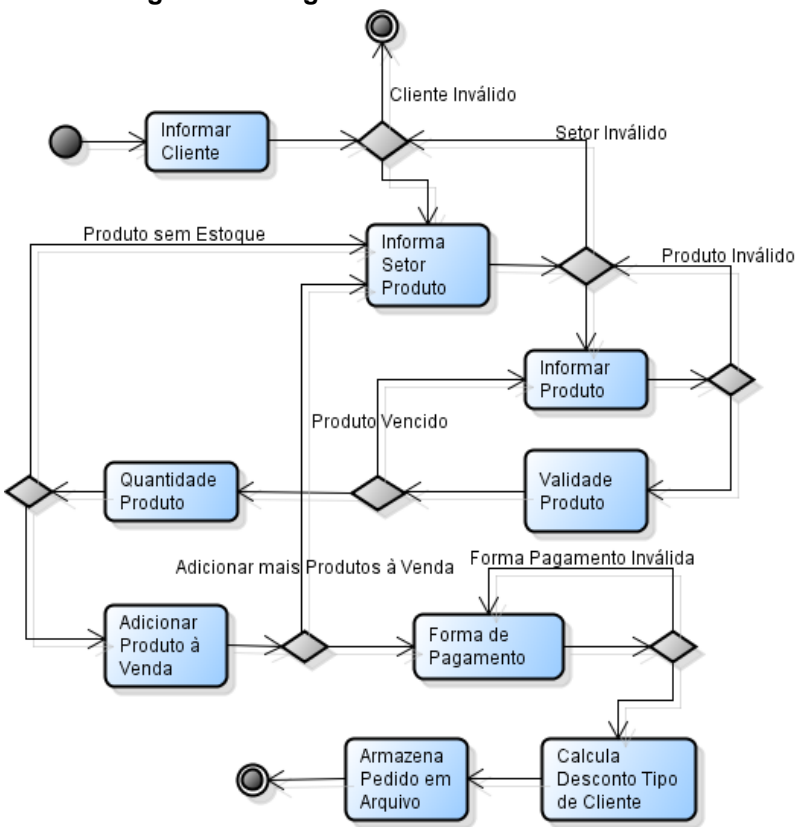
Nesse âmbito, os experimentos apresentados por Ferreira (2015), foram realizados tanto sobre a aplicação Mira ao Alvo quando em um segundo caso de estudo que consiste na implementação de um sistema de controle de vendas (pedidos de venda) tradicional. Em tempo, certamente que o entendimento dos experimentos

não é fundamental para a compreensão da aplicação do MCPON Preliminar feita por Ferreira, sendo aqui apenas relatos como conteúdo suplementar.

5.2.2.1 Programa de Controle de Vendas

Em linhas gerais, o programa consiste em uma implementação de um sistema de controle de vendas, o qual apresenta partes de um sistema tradicional no estilo *CRUD* (acrônimo de *Create, Retrieve, Update* e *Delete*) que permite criar, recuperar, atualizar e eliminar dados (SIMÃO *et al*., 2012b). Em suma, o escopo da aplicação pode ser vislumbrado no diagrama de atividades apresentado na Figura 61.

Figura 61: Programa de Controle de Vendas



Fonte: Ronszcka, 2012 via Diagrama de atividades em UML

Conforme ilustrado na Figura 61, o processo de venda se inicia com o informe do cliente que realizará o pedido. Uma vez escolhida e aprovada a venda para determinado cliente, devem ser informados os produtos que irão compor o pedido. O sistema possui validações quanto à existência de produtos e clientes. Ademais, o estoque de tais produtos é verificado. Se o produto escolhido para venda pertencer

ao setor de perecíveis, a sua data de validade é verificada. Na ocorrência de produtos vencidos, a venda não será permitida (RONSZCKA *et al*., 2015).

Depois de realizado o ciclo de informe de produtos, a venda poderá ser finalizada após a inserção da forma de pagamento, desconto e armazenamento do pedido. Na implementação desse sistema, existem apenas duas formas de pagamento possíveis, à Vista ou a Prazo. O cliente, em seu cadastro, possui uma informação sobre seu limite de crédito. Caso a forma de pagamento escolhida tenha sido a Prazo, o sistema verifica se o cliente tem permissão para efetuar a compra, confrontando o valor total do pedido com seu limite de crédito. Ainda, no cadastro do cliente há uma informação que lhe concede um tipo de classificação. Tal classificação é utilizada para a concessão de descontos especiais durante a finalização da venda. Para tanto, existe um total de 20 classes de clientes que dispõem de descontos que variam de uma faixa de 5% a 95%. Neste caso, para cada tipo de desconto será criada uma *Rule* correspondente. Desta forma, após a satisfação das *Premises*, a *Rule* em questão concederia o desconto do pedido de vendas para o cliente e, por fim, finalizaria sua venda (RONSZCKA *et al*., 2015).

De maneira a analisar a eficiência dos códigos gerados pelo sistema de compilação do PON, dois experimentos foram realizados no sistema em questão. O primeiro e o segundo experimento são representados pelo seguinte cenário, sendo tais experimentos distintos pela configuração do tipo de desconto concedido ao Cliente, no caso 1 e 20 respectivamente (FERREIRA, 2015).

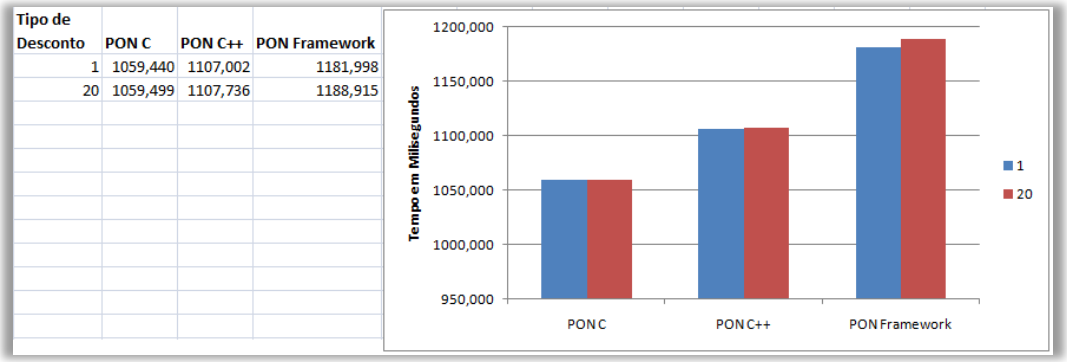
- Número de clientes: 1
- Número de produtos: 1
- Quantidade no estoque: 10.000
- Número de pedidos: 10.000
- Tipo de desconto do cliente: 1 e 20.

5.2.2.2 Teste de desempenho de execução

A Figura 62 apresenta os resultados obtidos após a execução dos cenários de testes. Neste experimento, o cenário de teste descrito acima foi executado por duas vezes. Em cada execução, o tipo de desconto do cliente foi configurado com o valor um e vinte respectivamente. Salienta-se que a escolha por estes valores se dá pela

diferença da quantidade de validações de expressões causais necessárias para avaliar o tipo de desconto do cliente (FERREIRA, 2015).

Figura 62: Resultado dos experimentos com Tecnologia LingPON 1.0: PON C (C específica a notificações), PON C++ (C++ específica a notificações), PON Framework (Framework PON C++ 2.0)



Tipo de Desconto	PON C	PON C++	PON Framework
1	1059,440	1107,002	1181,998
20	1059,499	1107,736	1188,915

Fonte: Ferreira, 2015

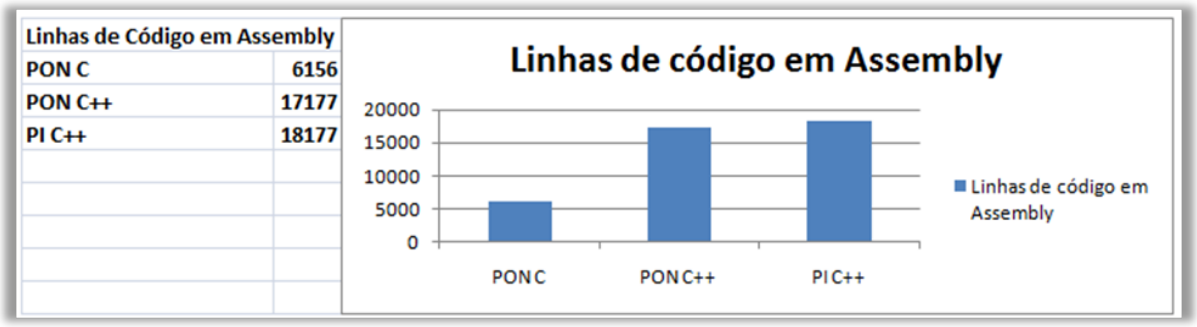
Conforme explicita o gráfico de resultados, neste experimento, a versão gerada pelo gerador de código para a versão em C específica *a notificações* foi a que apresentou os melhores resultados. Na sequência, a versão C++ *específica a notificações* obteve um desempenho superior quando comparada a versão do *Framework* PON C++ 2.0. Em contrapartida, a versão em C++ *específica a notificações*, novamente, apresentou o desempenho insatisfatório quando comparado a versão em C *específica a notificações*, mesmo comportamento este já considerado pelos experimentos realizados para a aplicação Mira ao Alvo (Seção 5.1.3.2).

Outra observação importante a ser destacada neste gráfico se dá pela variação de tempo mínima em cada versão quando o tipo de desconto do cliente é modificado, ou seja, o tipo de desconto do cliente varia de 1 para 20 e os resultados para o cliente configurado com o valor 1 com relação aos resultados para o cliente configurado com o valor 20 se apresentam com o tempo de processamento semelhantes. Neste experimento, é possível observar que as três versões se mantiveram com os valores constantes demonstrando que os resultados seguem os conceitos PON no que diz respeito à ausência de avaliação de expressões causais desnecessárias (FERREIRA, 2015).

5.2.2.3 Quantidade de linhas de código

A título de curiosidade experimental, outra medida realizada no experimento em questão, foi em termos de contagem das linhas de código em *Assembly* geradas para cada versão de código gerado. Esse experimento, visa validar as construções geradas para os códigos em C *específica a notificações* e C++ *específica a notificações* e realizar um comparativo com a contagem de linhas de código em *Assembly*. Ademais, o experimento apresenta, inclusive, uma versão do mesmo programa desenvolvida sob os princípios PI/POO em linguagem C++. Deste modo, o código em *Assembly* para cada uma das abordagens foi avaliado com a contagem de linhas de código com o intuito de validar a complexidade do código gerado. A Figura 63 apresenta os resultados para tal comparativo (FERREIRA, 2015).

Figura 63: Quantidade de linhas de código em *Assembly* - Sistema de Vendas via Tecnologia LingPON 1.0: PON C (*C específica a notificações*), *PON C++ (C++ específica a notificações), PON Framework (Framework PON C++ 2.0)*



Fonte: Ferreira, 2015

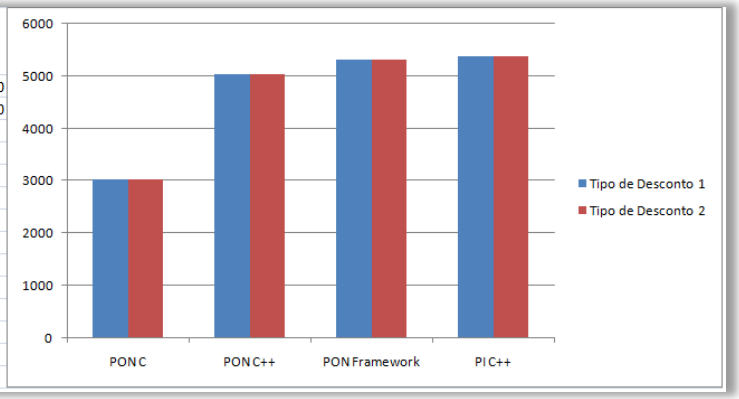
Como é possível observar na Figura 63, o código em *Assembly* gerado para o código em C *específica a notificações* foi o que apresentou a menor contagem de linhas de código. Isto valida, de certa forma, o fato do código gerado para a versão C *específica a notificações* apresentar o melhor tempo de execução quando comparado ao código em C++ *específica a notificações* e PI/POO C++. Já o código em *Assembly* gerado para C++ *específica a notificações* apresentou, de certa forma, uma contagem de linhas de código equivalente ao código gerado em PI/POO C++ (FERREIRA, 2015).

5.2.2.4 Teste de uso de memória

Assim como foi realizado o comparativo de utilização de memória na aplicação Mira ao Alvo (Seção 5.1.3.2), este mesmo comparativo foi realizado na aplicação Sistema de Vendas. A avaliação foi realizada ao longo da execução dos casos de teste e o maior consumo de memória foi capturado durante a execução. A Figura 64 ilustra os resultados obtidos para consumo de memória coletado durante os testes realizados.

Figura 64: Comparação do uso de memória entre os *targets* via Tecnologia LingPON 1.0: PON C (C específica a notificações), PON C++ (C++ específica a notificações), PON Framework (Framework PON C++ 2.0), PI C++ (Código puro em C++)

Tipo de Desconto	PON C	PON C++	PON Framework	PI C++
1	3024	5040	5320	5380
20	3024	5040	5320	5380



Fonte: Ferreira, 2015

Como é possível observar na Figura 64, a primeira coluna da tabela apresenta o tipo de desconto concedido ao cliente, enquanto a segunda coluna apresenta o uso de memória em Kilobytes para o programa compilado em C *específica a notificações*, a terceira coluna o total de memória utilizada pelo programa compilado em C++ *específica a notificações*, a quarta coluna apresenta o consumo de memória utilizada pelo *Framework* PON C++ 2.0 e, finalmente, a quinta coluna apresenta o total de memória utilizada pela aplicação desenvolvida sob o PI/POO C++ (FERREIRA, 2015).

Os resultados demonstram, assim como para a aplicação Mira ao Alvo, que o programa compilado em C *específica a notificações* foi o que atingiu o menor consumo de memória para executar os testes associados. Já o programa compilado em C++ *específica a notificações* apresentou um resultado um pouco inferior quando comparado ao uso de memória pelo *Framework PON C++ 2.0*. Ainda, é possível observar que a versão C++ *específica a notificações*, também, apresentou um

resultado um pouco inferior quando comparado ao uso de memória em PI/POO C++ (FERREIRA, 2015).

5.2.3 Considerações sobre a LingPON 1.0

Os resultados dos experimentos mostraram que a versão orientada a funções compilada para C *específica a notificações* foi a que apresentou melhores resultados em termos de desempenho, assim como também apresentou o menor uso de memória dentre as três versões comparadas. Além disso, o código *Assembly* gerado foi o que apresentou o menor número de linhas.

Desse modo, percebe-se que quanto mais próximo da linguagem de máquina a linguagem PON é compilada, menor é o seu consumo de memória, pois estruturas complexas e abstrações (*e.g.* classes) não são criadas, apenas o fluxo de execução, ocasionando assim, uma diminuição do consumo de memória (FERREIRA, 2015). O resultado completo de todos os experimentos e outras conclusões são apresentados em maiores detalhes em (FERREIRA, 2015).

Em suma, as conclusões de Ferreira (2015), mostram que a tecnologia LingPON é promissora, uma vez que em termos de facilidade de programação, a linguagem se apresentou como uma alternativa mais prática em comparação ao *Framework* PON C++ 2.0, abstraindo as características do PON em uma linguagem apropriada. A partir dessa linguagem, o desenvolvedor precisaria apenas conhecer a estrutura da mesma para criar programas no PON, tornando a concepção de programas no PON ainda mais em alto nível.

O sistema de compilação do PON, por sua vez, possibilitou a construção de materializações do PON que geram código mais otimizado e performante quando em comparação com os *frameworks* do PON, eliminando especialmente as estruturas de dados e objetos dinâmicos que se apresentavam como os principais gargalos da ferramenta. Esse objetivo foi atingido e os resultados apresentaram uma evolução considerável nesse âmbito. Em suma, a linguagem e o sistema de compilação convergiram para a melhoria do estado da técnica do paradigma como um todo.

Entretanto, a LingPON 1.0 ainda apresenta alguns problemas de falta de padronização, como a própria criação do novo bloco principal *main*, o qual uma vez utilizado, torna o código compilável apenas na plataforma definida a priori. Ademais, as versões de geração de código apresentam um ou outro defeito, em especial, nas

redundâncias estruturais presentes na execução do fluxo de notificações. Contudo, por ainda se tratar de uma tecnologia em amadurecimento, é natural que existam alguns defeitos, os quais foram melhorados na sequência, conforme explicam as próximas seções.

5.3 Tecnologia LINGPON 1.2

Os resultados promissores apresentados pela Tecnologia LingPON 1.0, instigaram a evolução do ferramental construído, particularmente à luz do método MCPON Preliminar e seu Sistemas de Compilação Preliminar. Para isso, foram ofertadas novas edições da disciplina “Linguagens e Compiladores” em 2015 e em 2016, com foco na Tecnologia LingPON e no método MCPON na parte prático-experimental, de forma a criar inclusive um fórum e laboratório de avanços e estímulos para a aplicação do método MCPON em outros desdobramentos do PON.

As edições de ambos os anos contaram com a colaboração de discentes e professores da UTFPR, além de membros do grupo de pesquisa do PON, tendo havido a participação do autor deste trabalho como tutor dos projetos realizados pelos colaboradores. No tocante aos resultados em si, das edições da disciplina em questão, há manuais e trabalhos, os quais estão apresentados com anexos em (SANTOS, 2017).

Nesses dois anos, estimulados inicialmente por esse fórum mencionado, surgiram algumas propostas de melhorias para a linguagem e principalmente para a geração de código para plataformas distintas. As evoluções da Tecnologia LingPON foram organizadas e compactadas em uma versão chamada de Tecnologia LingPON 1.2. Nesse âmbito, as subseções seguintes apresentam as particularidades dessa nova versão, salientando que houve evoluções na linguagem em si, nos geradores de códigos existentes, novos geradores de códigos e mesmo alguma melhoria no Sistema de Compilação Preliminar.

5.3.1 Melhorias sintáticas na linguagem

Em linhas gerais, com base nas colaborações de membros do grupo de pesquisa do PON, a versão da LingPON 1.2 foi sendo construída em prol de melhorar a questão de facilidade de programação, amadurecendo a linguagem para a

concepção de programas mais complexos, inclusive com novas estruturas e possibilidades, levando em conta os *feedbacks* de desenvolvedores que utilizaram as primeiras versões da linguagem.

Em sua dissertação de mestrado, Santos (2017) fez um levantamento das aplicações que haviam sido criadas em LingPON até aquele momento e constatou que haviam sido desenvolvidas aproximadamente 35 aplicações distintas[18]. Em geral, tais aplicações apresentam um escopo reduzido, pois visavam o estudo do desempenho do PON em cenários específicos de comparação. Nesse âmbito, dentre outros, ele construiu uma aplicação complexa usando a LingPON 1.0, nomeadamente uma aplicação de controle de Futebol de Robôs Simulados, para perceber vantagens e problemas da linguagem (SANTOS, 2017; SANTOS *et al*., 2017).

As investigações dele concluíram que o fato de apenas algumas aplicações PON terem sido desenvolvidas utilizando a LingPON estava relacionado inclusive a algumas limitações da linguagem, relativas ao encapsulamento de *Rules* internas aos *FBEs* e agregação entre *FBEs*. Como exemplo, imagina-se o próprio cenário de Futebol de Robôs, no qual era necessário reescrever *Rules* semelhantes, porém específicas para cada um dos jogadores do time. Tais restrições fazem com que o desenvolvedor tenha que reescrever, por diversas vezes, linhas de código muito semelhantes, como no caso de *Rules* similares (*e.g.* Chutar para o gol) para cada conjunto de instâncias de *FBEs* (*e.g.* Robôs), o que torna o processo oneroso e tendente a erros (SANTOS, 2017).

Visando solucionar tais limitações e permitir a criação de aplicações em PON com menor esforço, foram propostas, por Santos (2017), algumas melhorias na LingPON. A primeira proposta é a possibilidade de criar agregações entre *FBEs*, de modo que esses possam fazer referência a outros *FBEs*, aumentando o encapsulamento de *Attributes* e *Methods*. Outra questão é a proposta de criar agregações de *Rules* entre *FBEs*, de modo a permitir que um *FBE* possua *Rules* internas em sua estrutura, as quais são criadas pontualmente à medida que tais *FBEs* são instanciados (SANTOS, 2017). Dentre as principais evoluções para essa versão, é pertinente ressaltar a inclusão do conceito de Regras de Formação (*Formation*

[18] Muitas dessas aplicações, bem como outras subsequentes, estão relatadas nos trabalhos listados (quando não disponibilizados) na página de internet cujo *link* é o que segue: http://www.dainf.ct.utfpr.edu.br/~jeansimao/PON/PON.htm

Rules), realizado por Pordeus (2017), em um trabalho sinérgico ao de Santos. Outro trabalho sinérgico, este realizado por Schütz (2017), implementou o conceito de renotificações e não-notificações na estrutura da linguagem.

De modo a apresentar as melhorias propostas em maiores detalhes, as subseções seguintes exploram as nuances de cada qual.

5.3.1.1 *Formation Rules* (Regras de Formação)

De maneira geral, outro conceito proposto ao PON foi a extensão das *Formation Rules* (Regras de Formação) do Controle Orientado a Notificações (CON) (SIMÃO, 2001; SIMÃO; STADZISZ; KÜNZLE, 2003) para permitir a criação de *Rules* específicas, a partir da representação genérica de uma *Rule* (conforme Seção 3.5.5). Este conceito é útil quando o conhecimento causal de uma *Rule* é comum para diferentes conjuntos de instâncias de *FBEs*, ou seja, um conjunto de *Rules* específicas se diferencia apenas nas instâncias referenciadas.

Em suma, essa característica foi implementada na LingPON por meio de uma nova etapa de pré-compilação. Tal etapa transforma as *Formation Rules* em *Rules* tradicionais, de modo a serem compiladas normalmente pelas etapas de análise e síntese atuais.

Em maiores detalhes, para a adaptação do sistema de compilação com relação às *Formation Rules* foi necessário incluir novos componentes nas etapas de análise léxica e análise sintática na Etapa 1 do MCPON, novas instâncias na Etapa 2 do MCPON, particularmente no âmbito do Grafo PON e principalmente na Etapa 4, responsável pela geração de código.

Nesse âmbito, para a análise léxica foram adicionados novos *tokens* à LingPON (*i.e.*, *formRule* e *end_formRule*). Esses *tokens*, por sua vez, foram adicionados à *BNF*[19] que é utilizada na etapa de análise sintática. Em sequência, para o Grafo PON foi adicionado uma nova entidade instanciável, com o objetivo de identificar e mapear adequadamente as declarações de *Formation Rules* no grafo. Por fim, na etapa de geração de código, foi implementado um novo gerador de código específico, no qual a linguagem alvo é a própria LingPON, que neste caso é pré-

[19] A *BNF* da linguagem de programação LingPON 1.2 se encontra no Apêndice D.

processada para converter as *Formation Rules* em *Rules* tradicionais. O código final gerado por tal *target* deve ser novamente compilado, porém desta vez para alguma das outras opções disponíveis no Sistema de Compilação do PON (PORDEUS, 2017). A Figura 65 apresenta as etapas de compilação da linguagem PON após as adaptações realizadas.

Figura 65: Exemplo de *Formation Rules* no programa Sensores

Fonte: Autoria Própria

Conforme ilustra a Figura 65, as *Formation Rules* passam por uma etapa adicional de pré-compilação, fazendo o casamento (*matching*) de todas as instâncias de *FBEs* do tipo *Sensor* (linha 9) com a *formRule rlCheckSensorState* (linhas 12 a 20). Nesse processo, são criadas, basicamente, uma nova *Rule* especializada para as particularidades de cada instância do *FBE* em questão. Ademais, é importante ressaltar que para que uma *Rule* seja replicada para cada instância de um determinado *FBE*, é necessário que as *Premises* e *Instigations* façam referência ao nome do *FBE* ao invés da uma instância específica. No caso da *Formation Rule* do exemplo da Figura 65, é feita referência ao *FBE Sensor* (linha 17). Nesse caso, essa *Formation Rule* será replicada para cada combinação de instâncias do *FBE Sensor* com a instância do *FBE Alarm*.

5.3.1.2 Agregações de *Rules* – *FBE Rules*

De maneira geral, as agregações de *Rules* em *FBEs* é um conceito próximo ao das Regras de Formação. Tal conceito foi denominado anteriormente como *FBE Rules* e foi implementado inicialmente no *Framework* PON C++ 2.0 (RONSZCKA, 2012). Em suma, as *FBE Rules* consistem em um caso particular de *Formation Rules*, na qual a *Rule* está relacionada apenas a um determinado tipo de *FBE*, enquanto nas Regras de Formação, uma *Rule* pode estar relacionada a mais de um *FBE* (SANTOS, 2017).

Em outras palavras, a *Formation Rule* aparece como um elemento com escopo global na aplicação, criando relacionamentos *N* para *N* entre os *FBEs* participantes dessa regra especial. Os *FBE Rules*, por sua vez, possuem um escopo local ao *FBE* e todas as instâncias desse vão possuir obrigatoriamente uma instância da *Rule* em questão, formando um relacionamento *N* para 1. Nesse âmbito, o Código 37 apresenta um exemplo da estrutura sintática de um *FBE Rule* na LingPON.

Código 37: Exemplo de declaração de *FBE Rule* para o programa Sensores

```
1  fbe Sensor
2    attributes
3      string atState false
4    end_attributes
5    fbeRule rlCheckSensorState
6      condition
7        subcondition
8          premise prSensorState Sensor.atState == true
9        end_subcondition
10     end_condition
11     action
12       instigation inFireAlarm alarm.fire();
13     end_action
14   end_fbeRule
15 end_fbe
```

Fonte: Autoria Própria

Conforme apresenta o Código 37, as novas palavras reservadas *fbeRule* e *end_fbeRule* anunciam a criação de uma *Rule* específica para o *FBE Sensor*. É possível observar na linha 8 que a *Premise* em questão faz menção ao *FBE Sensor* e não a uma instância em particular. Assim, caso três instâncias do *FBE Sensor* sejam criadas e nomeadas respectivamente como “*sensor1*”, “*sensor2*” e “*sensor3*”, o pré-compilador criará três *Rules* no código pré-compilado, diferenciando-as apenas pela instância associada. Nesse âmbito, ao aplicar o novo conceito não é mais necessário

criar redundâncias das *Rules* manualmente, como era feito antes, onde era necessário criar explicitamente no código-fonte uma *Rule* para cada um dos três sensores.

5.3.1.3 Agregações entre *FBEs*

Outra questão foi possibilitar que a linguagem suportasse a agregação de *FBEs*, além dos tipos de dados primitivos (*i.e., boolean, char, integer, float e string*). Essa característica permite criar *FBEs* compostos, sem a necessidade de replicar múltiplos *Attributes*, um para cada característica de um Sensor, por exemplo, tornando o código mais coeso e sucinto. Nesse âmbito, o Código 38 apresenta um exemplo de agregação de *FBEs*.

Código 38: Exemplo de agregação de *FBEs* no programa Sensores

```
1  fbe Sector
2    attributes
3      Sensor sensor1
4      Sensor sensor2
5      Sensor sensor3
6    end_attributes
7    methods
8      method disarmSensors();
9    end_methods
10 end_fbe
```

Fonte: Autoria Própria

Conforme apresenta o Código 38, o *FBE Sector* é composto agora por três instâncias de *FBE Sensor*. Considerando que cada *Sensor* poderia ter *N Attributes* distintos, essa agregação eliminaria a necessidade de criar os mesmos *N Attributes* para cada um dos três elementos na formação do *FBE Sensor*. Essa redundância de definição de *FBEs* estava presente na versão 1.0 e era uma das principais críticas dos desenvolvedores que experimentaram a LingPON (SANTOS, 2017).

5.3.1.4 Propriedades reativas dos *Attributes*

De maneira geral, um conceito importante a ser utilizado nas aplicações PON diz respeito ao processo reativo de *Attributes*. Um *Attribute* pode apresentar três comportamentos distintos ao ter seu estado alterado:

- *Padrão*: notificar as *Premises* interessadas somente quando o estado de tal *Attribute* tiver sofrido alterações.
- *Renotificação*: notificar as *Premises* interessadas mesmo quando os estados de suas entidades colaboradoras não tenham sofrido alterações.
- *Não notificação*: não notificar as *Premises* interessadas mesmo que haja mudança nos *Attributes* relacionadas a estas.

Tal conceito do PON não havia sido implementado na versão 1.0 da LingPON sendo, portanto, uma contribuição adicional da versão 1.2 desenvolvida por Schütz. Nesse âmbito, o Código 39 apresenta a aplicação desse conceito na estrutura sintática da linguagem.

Código 39: Exemplo de implementação de reatividades dos *Attributes*

```
1  fbe Alarm
2    attributes
3      boolean atArmed false
4      boolean atFired false
5      boolean atBuzzer false
6    end_attributes
7    methods
8     method mtArm() (atArmed = true)
9     method mtDisarm() begin_method atArmed  = false<R>;
10                                   atBuzzer = false<N>;
11                                   atFired  = false<R>;
12                       end_method
13     method mtFire(atFired = true, atBuzzer = true)
14   end_methods
15 end_fbe
```

Fonte: Autoria Própria

O *Method mtDisarm()*, apresentado no Código 39, ajusta os *Attributes* do *FBE Alarm*, utilizando os conceitos de reatividade de *Attributes*. Nesse contexto, os caracteres entre os sinais de **< *e* >** denotam o tipo de reatividade do *Attribute*. Assim, **<*R*>** indica *renotificação* e **<*N*>** indica *não-notificação*. Caso não seja utilizado nenhum parâmetro após a atribuição de valores aos *Attributes*, o processo padrão de notificação é utilizado.

Em tempo, no exemplo do Código 39, as palavras-chave *begin_method* e *end_method* são utilizadas para delimitar o bloco de código que descreve os comandos do *Method*. Por conceito, o trecho de código-fonte será copiado, durante a compilação, diretamente para execução na linguagem alvo escolhida.

5.3.1.5 Considerações sobre as evoluções da LingPON 1.2

Em suma, as conclusões de Santos (2017) mostram que as evoluções da LingPON 1.2 são promissoras, uma vez que em termos de facilidade de programação, foi possível observar que a linguagem permitiu a criação de uma aplicação funcionalmente idêntica a desenvolvida a partir da versão 1.0, mas utilizando 61% menos linhas de código e 50% menos *tokens*. Isto influencia diretamente na velocidade de desenvolvimento de aplicações em PON por meio da LingPON (SANTOS, 2017).

Em suma, ao utilizar os novos conceitos apresentados na versão 1.2, é possível desenvolver aplicações PON com código-fonte mais enxuto. Desse modo, isso corrobora para o desenvolvedor encontrar ainda maior facilidade em adicionar ou alterar as entidades PON presentes no código-fonte. Sendo assim, a versão 1.2 da LingPON oferece maior concisão e facilidade de programação ao desenvolvedor que deseja programar para o PON, permitindo assim maior manutenibilidade e simplicidade de programação nesse paradigma.

5.3.2 Evoluções no sistema de compilação e nos geradores de código

Conforme apresentado, nos anos seguintes à construção da LingPON 1.0 surgiram tanto as melhorias para a estrutura sintática da linguagem quanto para os geradores de código. Nesse âmbito de evoluções nos geradores de código, cronologicamente, uma nova versão de geração de código foi criada, denominada LingPON *Estática* (*Static*). Tal versão diminuiu o tempo de processamento em relação a outras gerações de código em linguagem C++, mas dificultou a integração com outros códigos legados em C++. Essa dificuldade se deu principalmente por todo o código ser orientado a atributos e métodos estáticos do C++, o que inviabiliza a integração com código usual. Essa versão surgiu a partir da disciplina de 2015 e foi elaborada por Fernando Schütz com coautoria do autor desta tese (SCHÜTZ *et al.,* 2015; SCHÜTZ, 2018).

A disciplina de 2016, por sua vez, contou com a contribuição do aluno Eduardo Bilk de Athayde que construiu uma versão de geração de código chamada de LingPON *Espaço de Nomes* (*Namespaces*), no qual cada entidade do PON é tratado por um *espaço de nomes* (*namespace*) e não classe ou objetos visando eliminar

sobrecargas de OO e afins. A principal contribuição dessa versão foi buscar manter o baixo tempo de processamento da versão *estática*, propondo uma solução para os problemas de integração dessa última. A versão *estática*, mais precisamente, não possibilitava a extensibilidade do programa, uma vez que todas as classes são estáticas e não possibilitam a inclusão facilitada códigos não estáticos, inclusive aqueles de bibliotecas externas. Com base na versão *espaço de nomes*, foi possível integrar a instanciação de objetos e integração das chamadas de método OO com a execução do fluxo orientado a notificações (ATHAYDE e NEGRINI, 2016; ATHAYDE, 2017).

Ainda na disciplina de 2016, foram propostas, por Fabio Negrini, alterações tanto na Etapa 1 do MCPON, particularmente na *BNF* da linguagem LingPON 1.2 (Apêndice D) e na análise sintática, como também na Etapa 3 do MCPON, mais precisamente na subetapa de Criação de otimizadores genéricos independentes de *target*. As alterações implementadas foram: (a) reaproveitamento das entidades *Premises* por meio de nomes comuns, criando uma única instância de elementos *Premise* com as mesmas propriedades; (b) alterações na etapa de análise para permitir informar apenas o nome de uma *Premise* previamente declarada, sem a necessidade de detalhar as variáveis e operador de comparação da mesma; (c) validar a redefinição de *Premises* declaradas redundantemente no código, analisando a consistência dessas, as quais deveriam ser iguais caso apresentem o mesmo identificador; (d) vincular *Premises* sem identificador que apresentam a mesma condição de elementos previamente declarados. Em suma, tais otimizações buscaram eliminar a redundância de *Premises* com o mesmo propósito, buscando manter a essência de não-redundância do PON preservada (ATHAYDE e NEGRINI, 2016).

Paralelamente aos trabalhos voltados à software, no âmbito de hardware digital, Leonardo Faix Pordeus implementou uma nova versão de geração de código para código *AssemblyPON* voltado especificamente para a arquitetura *NOCA* (SCHÜTZ *et al.,* 2015; PORDEUS, 2017; KERSCHBAUMER, 2018). Esta implementação teve o objetivo de simplificar a tarefa de codificação direta em *AssemblyPON* e, principalmente, universalizar a implementação de programas no âmbito da LingPON.

De modo a apresentar as particularidades das versões distintas dos geradores de código propostas para a LingPON 1.2, a Seção 5.3.2.1 apresenta a versão de geração de código denominada geração *estática*. A Seção 5.3.2.2, por sua vez,

apresenta a versão de geração de código denominada geração com *espaço de nomes*. Por fim, a Seção 5.3.2.3 apresenta a geração de código para a arquitetura *NOCA*, mais precisamente em *AssemblyPON*.

Em tempo, salienta-se que as seções recém mencionadas no parágrafo acima apresentam conteúdo efetivamente técnico sobre as estratégias usadas dentro da construção de cada um dos geradores de código em questão, isto a título de enriquecimento informacional do tocante a essas implementações. Assim, a não leitura dessas seções em si não afetam em nada a compreensão de que esses geradores de código também foram outras demonstrações e aplicações de fases do MCPON preliminar.

5.3.2.1 Geração de código – Versão C++ com classes estáticas (*Static*)

De maneira geral, a proposta de geração de código para C++ denominada de LingPON *Estática* (*Static*) surgiu com o intuito de melhorar o desempenho do código PON escrito na linguagem C++, frente ao código PON escrito na linguagem C. A versão C++ do LingPON 1.0 teve uma diferença de desempenho bastante significativa em relação à versão em C *específica a notificações* (RONSZCKA *et al*., 2013; FERREIRA, 2015). Tal otimização é importante, pois a linguagem C++ (ao contrário da linguagem C) permite, por meio da abstração de classes OO, coesão e desacoplamento estrutural.

Neste contexto, uma tentativa de melhorar tal cenário seria refazer a geração em C++ utilizando conceitos das aqui chamadas classes estáticas, i.e., classes com atributos do tipo estático (*static*) e métodos estáticos *inline*. Além disso, o código foi refeito completamente, uma vez que a versão em C++ *específica a notificações* não apresentava otimizações em sua estrutura como um todo. O uso de métodos *inline* na linguagem C++ é tratado pelo compilador G++ como uma macro no momento da compilação. Sendo assim, as chamadas da função são eliminadas no programa executável, sendo substituídas pelo corpo (*i.e.,* instruções do algoritmo) da função declarada, eliminando assim o mecanismo de chamadas, empilhamento e retorno de funções. Tal característica é, naturalmente, positiva para tal proposta de execução orientada a chamadas de métodos no tocante ao código gerado, uma vez que cada notificação é representada por uma chamada de método em termos de codificação, mas não em termos de execução (SCHÜTZ *et al.,* 2015).

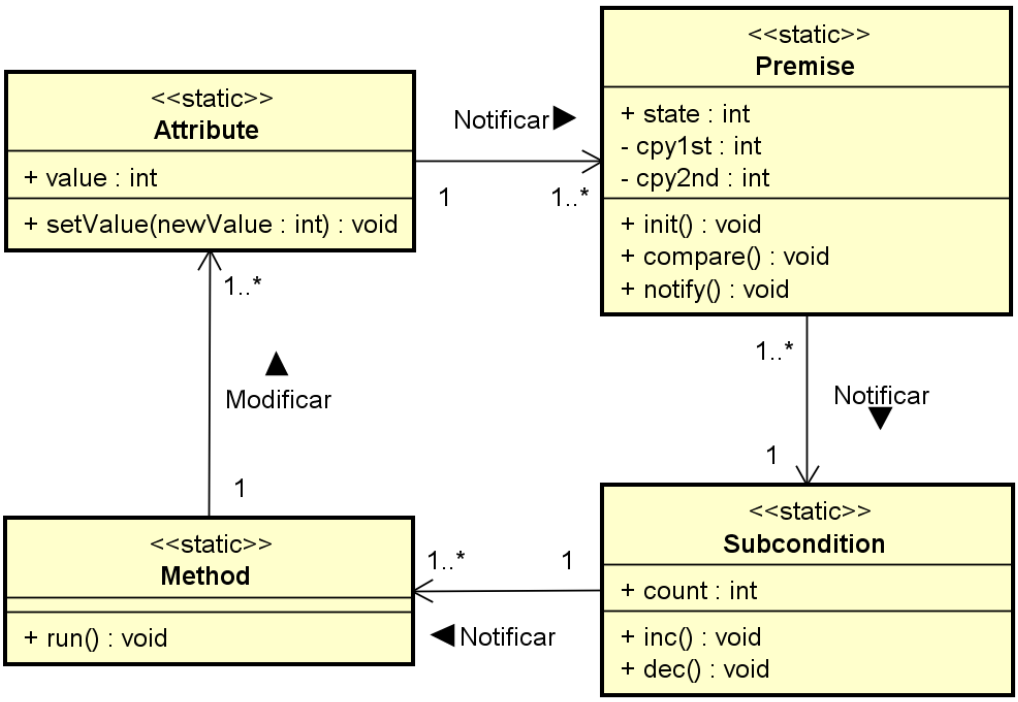
Outra característica fundamental foi eliminar a utilização de atributos e métodos de instância, transformando todos eles em elementos estáticos (*static*). Dessa forma, a estrutura geral do programa gerado não utiliza variáveis nem instâncias de classe não estática (i.e., classes usuais). Ademais, o espaço de armazenamento para os objetos é alocado quando o programa inicia e desalocado quando o programa termina, utilizando apenas uma instância do objeto. Este processo elimina o tempo de alocação dinâmica durante a execução do programa. Assim, cada uma das entidades PON proposta para essa versão foi compilada no formato de classe estática (SCHÜTZ *et al.,* 2015).

As classes do modelo geral do PON se apresentam da seguinte forma, na versão *estática* da LingPON:

- *Attribute*: possui um atributo estático para representar o valor e um método *inline* estático para a atribuição de novos valores a tal variável.
- *Premise*: possui dois métodos estáticos. Um método controla o processo de comparação entre os *Attributes* e sua consequente notificação, enquanto o outro método controla o estado inicial da entidade.
- *Subcondition*: possui dois métodos estáticos de controle para incrementar e decrementar a quantidade de *Premises* aprovadas durante o processo de inferência.
- *Method*: possui basicamente a implementação dos comandos designados, por meio de um método estático.

A simplificação da estrutura em quatro classes, conforme listado, possibilitou uma execução mais enxuta e otimizada do código-fonte PON resultante. A Figura 66 apresenta o diagrama de classes UML da implementação *estática* da LingPON.

Figura 66: Estrutura dos elementos da versão *estática*

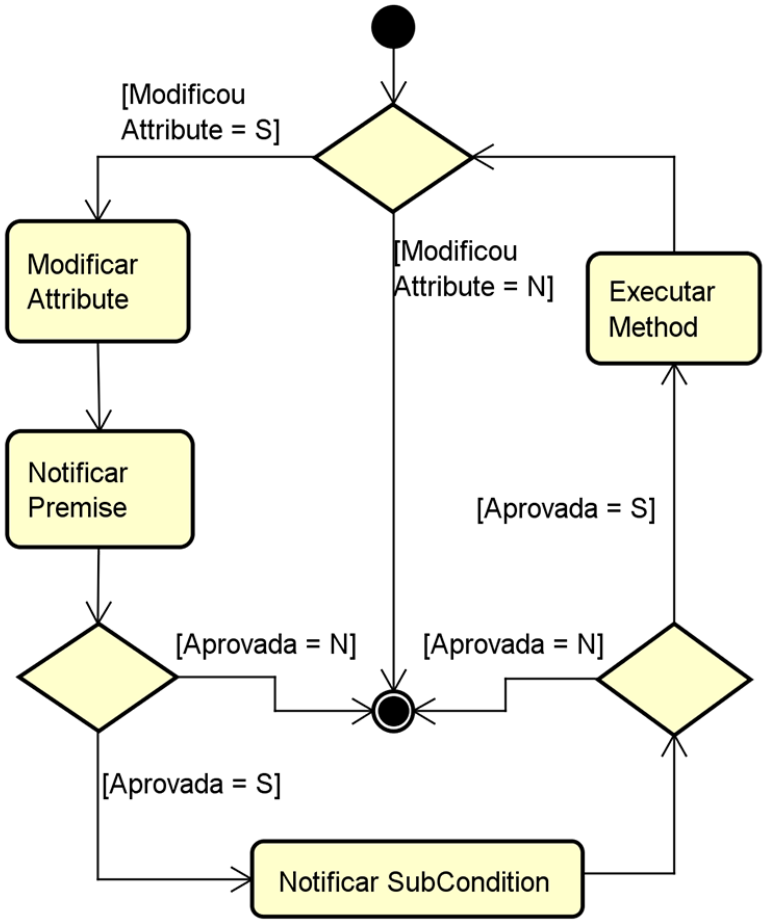


Fonte: Schütz *et al*., 2015 via Diagrama de classes em UML

Como pode ser observado na Figura 66, o código fonte gerado não implementa classes para os elementos *Rule*, *Action* e *Instigation* da estrutura do modelo conceitual do PON. Tais supressões foram realizadas visando otimizações e melhoria no desempenho das aplicações geradas. A implementação da *Rule* foi suprimida pelo fato de seus conceitos estarem abstraídos na classe *Subcondition*. Ainda, tanto a *Action* quanto a *Instigation* apresentam características uteis particularmente (ainda que não apenas) para execução paralela e *multicore*. Assim, como a versão *estática* da LingPON foi desenvolvida para ambientes monoprocessados, sua implementação foi suprimida.

Ademais, a Figura 67 apresenta um diagrama de atividades da UML que ilustra o fluxo de execução otimizado da cadeia de notificações na *LingPON Estática*.

Figura 67: Fluxo de execução da versão para C++ Estática



Fonte: Schütz *et al*., 2015 via Diagrama de atividades em UML

Conforme apresenta a Figura 67, o fluxo se dá por meio da mudança de estado (*value*) de um *Attribute*, advindo de uma chamada ao método *setValue*() em dado momento. Ao ter seu estado alterado, tal *Attribute* notifica as *Premises* relacionadas, por meio da execução do método *notify*() de cada classe *Premise*. Tal método ajusta o valor das variáveis que representam os valores dos *Attributes* que pertencem à avaliação causal, e executa o método *compare*(). Este método executa a avaliação causal propriamente dita e, no caso de mudanças de estado nas Premises, estas notificarão as *Subconditions* interessadas.

Neste contexto, caso o estado (*state*) da *Premise* seja alterado de falso para verdadeiro, o método *inc*() da *Subcondition* pertinente é chamado; no caso do estado passar de verdadeiro para falso, o método *dec*() é executado. Tais métodos incrementam e decrementam a quantidade de *Premises* aprovadas que estão relacionadas à *Subcondition* pertinente. Ao atingir o número de aprovações necessárias, a *Subcondition* atua como uma *Rule*, instigando a execução de um

Method. Tal processo se dá pela execução do método *run*(), presente em cada uma das classes *Method*. Os comandos de tal método podem alterar um *Attribute*, fazendo com que um novo fluxo de notificação seja executado. Caso contrário, o processo é encerrado.

De modo a apresentar as características dessa geração de código, em especial, os códigos 40 a 43 apresentam o resultado da compilação do programa Sensores.

Código 40: Trecho de código gerado para um Attribute na compilação para C++ com Classes Estáticas

```
1  class sensorPresence1_atState {
2    public:
3      static bool value;
4      static inline bool setValue(bool newValue) {
5        if (sensorPresence1::value != newValue) {
6          sensorPresence1::value = newValue;
7          prSensorPresence1::notify_sensorPresence1_atState(newValue);
8        }
9      }
10 };
```

Fonte: Autoria Própria

Conforme apresenta o Código 40, o trecho de código gerado para um *Attribute* tem uma estrutura inteiramente estática e otimizada. Na linha 3, particularmente, a classe possui apenas um *Attribute* que armazena o estado do elemento em questão. As linhas de 4 a 9, por sua vez, apresentam o método para a mudança do estado do *Attribute*, validando inicialmente a mudança efetiva do estado, para só então notificar as *Premises* interessadas.

É importante observar que no programa compilado na versão LingPON *Estática*, é criada uma nova classe para cada instância de um *Attribute*, uma vez que tal classe representa o estado individual da entidade, bem como as notificações pontuais para a(s) instância(s) da(s) *Premise(s)* relacionada(s).

Nesse âmbito, as classes geradas são nomeadas de acordo com a implementação definida pelo programador no código PON, mantendo um padrão de nomenclatura que nesse caso segue o nome da instância do *FBE* e o nome do *Attribute* criado (*i.e., sensorPresence1* e *atState*).

Assim como os *Attributes*, para as instâncias das *Premises* do programa compilado também são geradas classes pontuais que representam as particularidades

de cada qual. Nesse âmbito, o Código 41 apresenta um exemplo de uma *Premise* do programa Sensores.

Código 41: Trecho de código gerado para uma Premise na compilação para C++ com Classes Estáticas

```
1  class prSensorPresence1 {
2    public:
3      static bool state;
4      static bool cpy1st_att;
5      static bool cpy2nd_att;
6      static inline void init() {
7        prSensorPresence1::cpy1st_att = false;
8        prSensorPresence1::cpy2st_att = true;
9        if (prSensorPresence1::cpy1st_att ==
10           prSensorPresence1::cpy2st_att) {
11         prSensorPresence1::state = true;
12         scIntruderIdentified.inc();
13       }
14     }
15     static inline void notify_sensorPresence1_atState(bool newValue) {
16       prSensorPresence1::cpy1st_att = newValue;
17       prSensorPresence1::compare();
18     }
19     static inline void compare() {
20       if (prSensorPresence1::cpy1st_att ==
21           prSensorPresence1::cpy2st_att) {
22         if (prSensorPresence1::state == false) {
23           prSensorPresence1::state = true;
24           scIntruderIdentified.inc();
25         }
26       } else {
27         if (prSensorPresence1::state == true) {
28           prSensorPresence1::state = false;
29           scIntruderIdentified.dec();
30         }
31       }
32     }
33 };
```

Fonte: Autoria Própria

Conforme apresenta o Código 41, uma *Premise* apresenta uma estrutura padrão, composta por alguns elementos específicos e outros genéricos que são comuns entre elas. Mais precisamente, cada *Premise* possui um *Attribute* que armazena seu estado interno (linha 3) e dois *Attributes* que armazenam uma cópia do último valor conhecido de cada um dos elementos comparáveis (linhas 4 e 5).

O fato de armazenar tais valores na essência da entidade permite que as *Premises* tenham um comportamento totalmente reativo, evitando buscas por valores em outras entidades do paradigma. Ademais, tal entidade é composta por um método de inicialização (linhas 6 a 14) que permite em tempo de montagem do programa, definir o estado de cada entidade.

Nesse exemplo, em particular, a *Premise* possui uma comparação entre um *Attribute* com valores dinâmicos (*atState*) e uma constante de valor fixo (*true*). Nesse caso é criado apenas um método de notificação entre o primeiro elemento e a *Premise* em questão (linhas 15 a 18). As linhas 19 a 32, representam a parte dinâmica da entidade, na qual o corpo da comparação é definido baseado nas particularidades da mesma.

Ainda, são definidas as conexões por notificação com as *Subconditions* a serem notificadas em caso de aprovação (linha 24) ou reprovação (linha 29) da *Premise* em questão. A relação por notificação entre uma *Premise* e suas *Subconditions* relacionadas está presente nos métodos *inc* e *dec*, os quais são apresentados no Código 42.

Código 42: Trecho de código gerado para uma *Subcondition* na compilação para C++ com Classes Estáticas

```
1  class scIntruderIdentified {
2    public:
3      static int count;
4      static inline void inc() {
5        scIntruderIdentified::count++;
6        if (scIntruderIdentified::count == 2) {
7          alarm1_mtRingTheBell::run();
8        }
9      }
10     static inline void dec() {
11       scIntruderIdentified::count--;
12     }
13 };
```

Fonte: Autoria Própria

Conforme apresenta o Código 42, a estrutura de uma *Subcondition* é composta basicamente por um *Attribute* contador (linha 3) que armazena o número de *Premises* aprovadas e dois métodos para incrementar e decrementar tal *Attribute*. Conforme o contador atinja o número de *Premises* necessárias para a aprovação da *Subcondition* em questão (linha 6), o *Method* relacionado é instigado a executar (linha 7). O trecho de código do Código 43, apresenta a essência de em *Method*.

Código 43: Trecho de código gerado para um *Method* na compilação para C++ com Classes Estáticas

```
1  class alarm1_mtRingTheBell {
2    public:
3      static inline void run() {
4        alarm1_atTimer::setValue(60);
5      }
6  };
```

Fonte: Autoria Própria

Conforme apresenta o Código 43, o *Method* em questão se resume a atribuição do valor 60 para o *Attribute atTimer*. Esse, por sua vez, encadearia notificações para outras *Rules* e, assim, o fluxo de notificações da versão compilada da LingPON *Estática* está completo, retomando as notificações para a entidade *Attribute*.

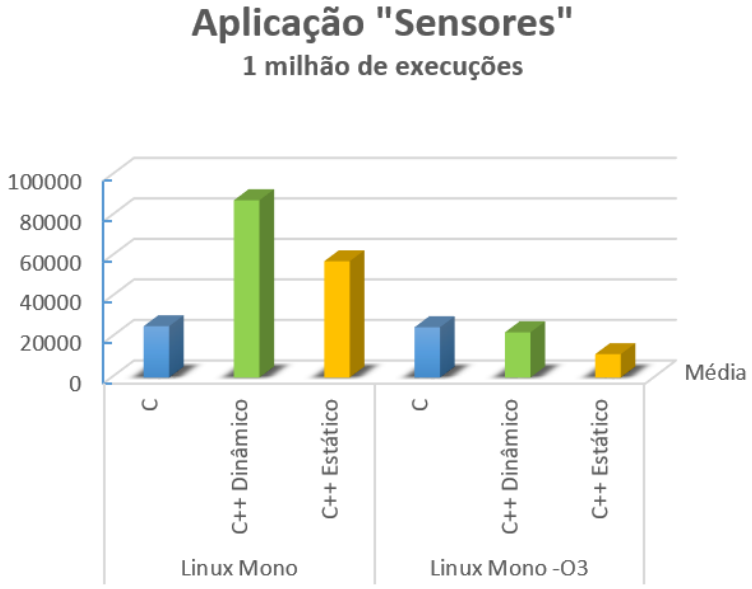
É importante salientar que todas as classes possuem sua essência formada por atributos e métodos estáticos, dessa forma toda a execução acontece inteiramente na pilha de execução do programa. Ademais, existe apenas uma única cópia de cada elemento, acessíveis por todas as entidades de forma pública e global, evitando assim alocações dinâmicas e gargalos proporcionados pelo mal-uso dessas.

Ainda, experimentos foram realizados de modo a verificar a eficiência dessa versão em comparação as demais versões. Nesse experimento foram realizadas um milhão de alterações nos *Attributes*, de modo a ativar as *Rules* pertinentes. Nesse âmbito, de modo a validar a aplicação, três versões compiladas foram testadas, sendo estas PON em linguagem C *específica a notificações* (Tecnologia LingPON 1.0), linguagem C++ *específica a notificações* (Tecnologia LingPON 1.0) e na versão em linguagem C++ *estática* (SCHÜTZ *et al.,* 2015).

Nesse sentido, a Figura 68 apresenta os resultados do experimento, no qual o gráfico foi dividido em duas partes. A primeira parte (lado esquerdo) apresenta o resultado para compilação monoprocessada sem parâmetros de otimização para o compilador G++. No primeiro experimento, a versão de código C *específico a notificações* apresentou os melhores resultados, mesmo em comparação com a nova versão de código C++ *estático* (a notificações certamente). Entretanto, a versão de código C++ *estático* apresentou-se mais eficiente que a versão de código C++ *específico a notificações* (SCHÜTZ *et al.,* 2015).

O segundo experimento (lado direito) levou em conta a ativação da otimização O3 no compilador G++. Com ela foi possível observar uma melhora significativa das versões de código C++ *específico a notificações* e versão de código C++ *estática* em comparação com a versão C *específica a notificações* (SCHÜTZ *et al.,* 2015).

Figura 68: Experimento de tempo de execução entre versões da LingPON



Fonte: Schütz *et al.*, 2015

Em suma, é possível observar que a nova versão *estática* apresentou resultados satisfatórios, comprovando que a utilização de elementos estáticos em sua essência contribui para um aumento de desempenho na execução de programas escritos em PON. Ademais, testes adicionais com essa versão foram propostos e apresentados em outras aplicações e são apresentados em (SCHÜTZ, 2019).

5.3.2.2 Geração de código – Versão Espaço de Nomes (*Namespaces*)

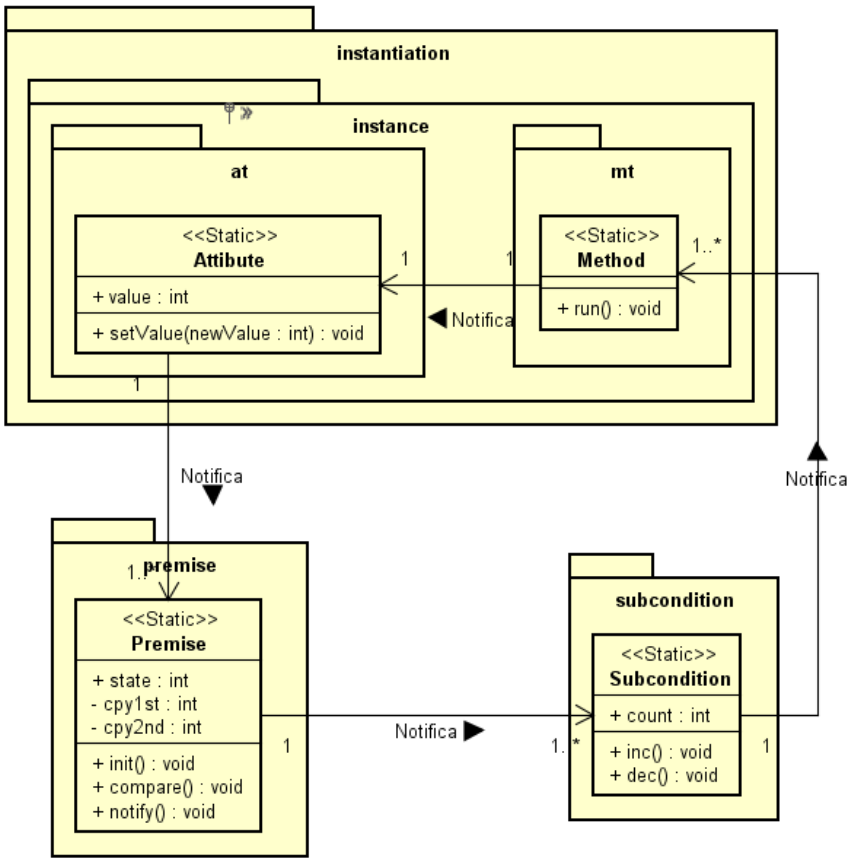
Conforme apresentado na seção anterior, a compilação da versão LingPON *Estática* apresentou melhorias significativas na organização do código gerado, bem como no tocante a desempenho. Contudo, ela trouxe um ônus para a extensibilidade do código PON, uma vez que todas as classes geradas são estáticas.

Nesse contexto, foi proposta uma versão de geração de código sem a presença de elementos estáticos, porém com o uso de *espaço de nomes*, os quais permitem criar variáveis globais internamente no escopo de seus espaços de nome. Dessa forma, não foi necessário criar classes e instâncias dinâmicas de objetos, mantendo a essência da organização do código bastante próxima ao resultado da versão *estática*. Outra vantagem da versão *namespaces* é a possibilidade de integração de código de terceiros e inclusão de bibliotecas externas na estrutura de execução das *Rules* PON (ATHAYDE e NEGRINI, 2016).

Isto considerado, *espaço de nomes* em C++ podem se referir a variáveis, funções, estruturas, enumerações, classes e membros de classes ou estruturas. Ademais, *espaço de nomes* podem ajudar a tratar ambiguidade em nomes de entidades como variáveis e funções por meio do escopo de cada *espaço de nomes* como a própria designação “espaço de nomes” indica. De fato, à medida que uma aplicação cresce, a necessidade de gerenciar ambiguidade aumenta (ATHAYDE e NEGRINI, 2016).

Em linhas gerais, a implementação da versão *espaço de nomes* seguiu a mesma estrutura organizacional e lógica da versão *estática*. Contudo ao invés de gerar um arquivo para cada entidade, agrupou-se todas as *Premises* em um único arquivo chamado de *Premises.h*, todas as *Subconditions* em um único arquivo chamado *Subconditions.h* e cada instância de *FBE*, contendo seus respectivos *Attributes* e *Rules*, em um único arquivo chamado *Instantiations*.h. Nesse âmbito, a Figura 69 apresenta a estrutura organizacional e o fluxo de execução da cadeia de notificações na LingPON *Espaço de Nomes* (ATHAYDE e NEGRINI, 2016).

Figura 69: Estrutura da LingPON *Espaço de Nomes*



Fonte: Athayde e Negrini, 2016 via Diagrama de classes em UML

Conforme ilustra a Figura 69, o fluxo de notificações e a estrutura lógica das classes da versão *estática* foi mantida, com a novidade da inclusão de pacotes representando os *espaços de nomes* e, obviamente, o não extensivo do uso de código oriundo de atributos e métodos estáticos. O primeiro pacote agrupa *Attributes* e *Rules* em um *espaço de nomes* que representa o nome da instância do *FBE*. Ademais, todas as instâncias de *FBEs* do programa compilado são agrupadas em um *espaço de nomes* único e geral denominado de *instantiation*. Por sua vez, todas as *Premises* e *Subconditions* da aplicação são agrupadas em *espaço de nomes* denominados de *premise* e *subcondition*, respectivamente.

De modo a apresentar as características dessa geração de código, em especial, os códigos 44 a 46 apresentam o resultado da compilação do programa Sensores.

Código 44: Trecho de código gerado para instâncias de FBE na LingPON Espaço de Nomes

```
1  namespace instantiation {
2    namespace alarm1 {
3      namespace at {
4        namespace atOn {
5          extern bool value;
6          extern void setValue(bool newValue);
7        }
8        namespace atTimer {
9          extern int value;
10         extern void setValue(int newValue);
11       }
12     }
13     namespace mt {
14       extern bool mtRingTheBell();
15     }
16   }
17   namespace sensorPresence1 {
18     namespace at {
19       namespace atState {
20         extern bool value;
21         extern void setValue(bool newValue);
22       }
23     }
24   }
25 }
```

Fonte: Autoria Própria

Conforme apresenta o Código 44, o código substitui totalmente a utilização de classes e passa a ser baseado em *espaço de nomes*. Cada *espaço de nomes* possui relevância global dentro de seu escopo. Para isso, a linha 1 apresenta o *espaço de nomes* global para o programa todo denominado de *instantiation*.

Ademais, nesse espaço de nome em questão, conforme apresenta o trecho de código das linhas 2 e 17, acomoda-se todas as instâncias de *FBEs* do programa. Ademais, a palavra *extern* é utilizada para declarar a existência dos elementos globais (*i.e.* variáveis e métodos) no cabeçalho do programa (arquivo .h), para que cada arquivo fonte (arquivo .cpp) conheça sobre a existência desses elementos globais e possam acessá-los normalmente. Entretanto, as definições concretas estão presentes em um arquivo fonte específico e único. Seguindo a mesma linha de declarações globais em *espaço de nomes*, o arquivo de cabeçalho contendo as *Premises* é apresentado no Código 45.

Conforme apresenta o Código 45, as *Premises* também possuem um *espaço de nomes* de escopo global para o programa denominado de *premise*. Nesse *espaço de nomes* estão acomodadas todas as *Premises* do programa. Conforme apresenta o trecho de código, as declarações de variáveis e métodos seguem o mesmo padrão adotado pela implementação da versão *estática*.

Código 45: Trecho de código gerado para *Premises* na LingPON Espaço de Nomes

```
1  namespace premise {
2    namespace prSensorPresence1 {
3      extern bool atState;
4      extern bool cpy1st, cpy2nd;
5      extern void init();
6      extern void notify_sensorPresence1_atState(bool newValue);
7      extern void compare();
8    }
9  }
```

Fonte: Autoria Própria

De modo a apresentar o código fonte de definição dos elementos desta versão, o Código 46 apresenta a definição do espaço de nome para as *Subconditions*.

Código 46: Trecho de código gerado para *Subconditions* na LingPON Espaço de Nomes

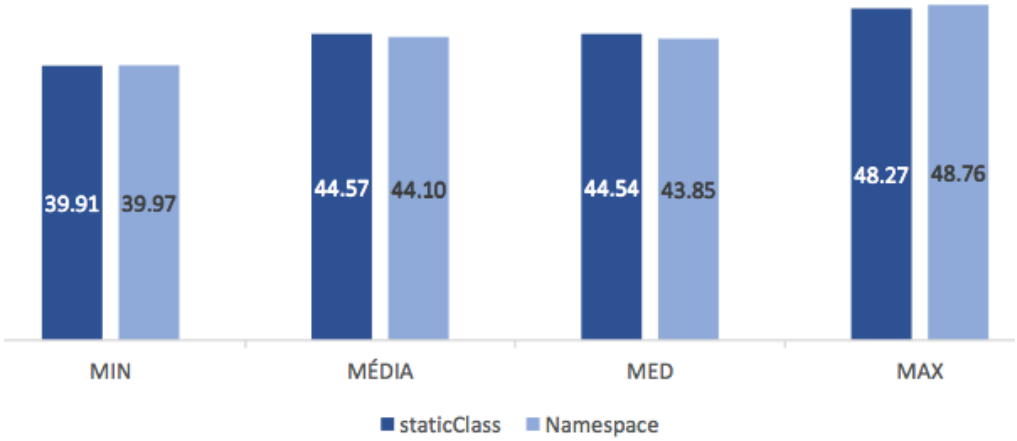
```
1  namespace subCondition {
2    namespace scIntruderIdentified {
3      int count;
4      void inc() {
5        count++;
6        if (count == 2) {
7          instantiation::alarm1::mt::mtRingTheBell();
8        }
9      }
10     void dec() {
11       count--;
12     }
13   }
14 }
```

Fonte: Autoria Própria

Conforme apresenta o Código 46, a implementação da lógica da definição do código nessa versão segue o mesmo padrão adotado na versão *estática*. É possível observar, na linha 7, como se dá o acesso global aos demais elementos do programa, os quais são acessíveis por meio dos *namespaces* globais (*e.g. instantiation*).

De maneira a investigar o impacto em termos de desempenho dessa versão em comparação com a versão *estática*, um experimento foi realizado e apresentado no Anexo E em (SANTOS, 2017), com o aplicativo portão eletrônico. Considerou-se o tempo, em segundos, de um milhão de acionamentos de portão (aprovação da *Rule*). Ademais, a compilação no G++ foi realizada em otimização O3 em um ambiente Linux, para ambas as versões (ATHAYDE e NEGRINI, 2016). O experimento em si contou com oito mil amostras do resultado o qual é apresentado no gráfico ilustrado na Figura 70.

Figura 70: Gráfico comparativo entre versão *estática* e versão *espaço de nomes*



Fonte: Athayde e Negrini, 2016

Conforme ilustra a Figura 70, os resultados se mostraram satisfatórios, uma vez que não reduziram o desempenho em relação à versão de código específico do PON em C++ *estático*. Nesse contexto, tal implementação trouxe um impacto positivo, ao passo que resolve aquela deficiência de não possibilitar a extensibilidade do código, presente na versão do código específico PON em C++ *estático* (ATHAYDE e NEGRINI, 2016).

Em linhas gerais, o código não faz uso de classes, apesar de ser um código em C++. Entretanto, os espaços de nome substituem, em partes, a essência estrutural das classes e, principalmente, aninham um espaço de nome em outro, tornando o código mais organizado e principalmente reduzindo o número de classes e arquivos.

Nesse âmbito, a implementação da LingPON *Espaço de Nomes* permitiu uma maior versatilidade na geração para C++ *específica a notificações*, sem comprometer os ganhos de desempenho presentes na versão *estática*.

5.3.2.3 Geração de código – *AssemblyPON*

Em linhas gerais, na dissertação de Pordeus (2017) foram apresentadas algumas adaptações na LingPON para gerar *AssemblyPON*, útil tanto à *NOCA* quanto ao *NOCASim* (respectivamente arquitetura computacional própria ao PON e seu simulador explicados na Seção 3.4.2). Tais adaptações tiveram como objetivo facilitar o desenvolvimento de aplicações para a *NOCA*, pois a linguagem PON fornece um nível de abstração superior quando comparado ao desenvolvimento de aplicações por meio de *AssemblyPON* (PORDEUS, 2017). Basicamente, as adaptações em questão ocorreram na etapa de síntese, aproveitando totalmente as análises da linguagem. Isto é, o código todo foi gerado a partir do Grafo PON, não necessitando alterações na estrutura da linguagem.

De modo a apresentar um exemplo de código alvo em *AssemblyPON* gerado pelo gerador de código especializado, o Código 47 apresenta um trecho deste código, no qual os elementos PON: *Attributes*, *Premises*, *Conditions* e *Rules* escritos em linguagem PON, são representados por meio de suas respectivas instruções segundo a *ISA* da *NOCA*: *ATTRIBUTE-DECL*, *PREMISE-OP*, *CONDITION-OP*, *METHOD-OP* e *METHOD-VN-OP*.

Código 47: *AssemblyPON* da aplicação *Counter*

```
1  #define VN.counter.mtEnd 4029c end_method
2
3  counter.count: ATTRIBUTE-
4  DECL,,,N,,2,0,PrCounterEquals5,PrCounterLess5;
5
6  PrCounterEquals5:
7  PREMISE-OP,,,,,,AV,==,1,counter.count,5,CdCounterEquals50;
8
9  PrCounterLess5:
10 PREMISE-OP,,,R,,,AV,<,1,counter.count,5,CdCounterLess50;
11
12 @c00020
13 CdCounterEquals50:
14 CONDITION-OP,,,,,0,!!,2,PrCounterEquals5,,counter.mtEnd,counter.mtRst;
15
16 CdCounterLess50:
17 CONDITION-OP,,,R,,0,!!,1,PrCounterLess5,,counter.mtInc;
18
```

```
19 @c00050
20 counter.mtInc:
21 METHOD-
22 OP,,,,,AV,INC,IC=0,0,CdCounterLess50,counter.count,,counter.count;
23
24 counter.mtRst:
25 METHOD-OP,,,,,VV,=,IC=0,0,CdCounterEquals50,0,,counter.count;
26
27 counter.mtEnd: METHOD-VN-OP,VN.counter.mtEnd,0;
```

Fonte: Pordeus, 2017

Em suma, a geração para *NOCA* apresenta duas características importantes. A primeira é a possibilidade de desenvolvedores criarem aplicações que executariam na estrutura da *NOCA* com pouco conhecimento prévio da arquitetura da *NOCA* e do *AssemblyPON* necessário para tal. A segunda é universalidade da linguagem e compilador para a geração de código em múltiplas plataformas.

5.4 LINGPON HD

Em sua tese de doutorado, Kerschbaumer (2018) apresenta uma implementação do PON, na qual todos os elementos desse paradigma são modelados em blocos de lógica reconfigurável, utilizando a linguagem VHDL, inspirado à luz de uma implementação preliminar e limitada proposta por Witt *et al.* (2011) e outros trabalhos correlatos (SIMÃO *et al.*, 2012e; PORDEUS *et al.*, 2016). Essa nova implementação de PON para hardware digital, chamada de PON-HD 1.0, foi desenvolvida para facilitar a síntese em *FPGA* (conforme detalhado na Seção 3.4.2) (KERSCHBAUMER, 2018; KERSCHBAUMER *et al*., 2018).

Nesse âmbito, inicialmente Kerschbaumer, em sua qualificação de doutorado (KERSCHBAUMER, 2017), fez uma nova linguagem de programação para possibilitar que essa gerasse código VHDL com os blocos orientados a notificações, definidos na especificação do PON-HD 1.0. Essa versão da linguagem e gerador de código foi denominada de LingPON HD 1.0, uma vez que ela apresenta um dialeto paralelo ao da LingPON original. Na LingPON HD 1.0 são definidos outros elementos peculiares ao VHDL, tais como endereços de memória e portas de entrada e saída. Com essa nova versão da LingPON é possível gerar código VHDL para FPGA diretamente de um programa PON escrito em linguagem em alto nível. A título de exemplo, um programa escrito em LingPON HD 1.0 é apresentado no Código 48 (KERSCHBAUMER, 2017).

Código 48: Exemplo de programa em LingPON HD 1.0

```
1  entity Contador
2    //attributes
3    attribute integer atCounter 0 out
4    attribute integer atOn 0 in
5    //end_attributes
6    //methods
7    method mtIncrement atCounter = atCounter + 1
8    //end_methods
9  //end_fbe
10 //Rule rlCont
11   //Condition cdCont
12   premise prLess atCounter < 10
13   premise prOn atOn == 1
14   //end_condition
15   //action acCont
16     rule rlCont mtIncrement prOn and prLess
17   //end_action
18 //end_rule
```

Fonte: Kerschbaumer, 2017

Conforme apresenta o Código 48, a linguagem, em geral, simplifica, porém deforma a estrutura original da linguagem, mantendo apenas algumas das palavras reservadas fundamentais (*i.e., entity*, *attribute*, *method*, *premise*, *rule*). Também é possível observar as operações realizadas por esses elementos e os valores envolvidos, como a *Premise prLess*, a qual verifica se o *Attribute atCounter* é menor que a constante 10. Ainda, é possível observar as interconexões entre os elementos, como por exemplo, na *Rule rlCont*, a qual executa o *Method mtIncrement* quando as *Premises prOn* e *prLess* forem verdadeiras (KERSCHBAUMER, 2017).

É importante ressaltar que nessa implementação específica, Kerschbaumer não utilizou o método MCPON para a construção de sua linguagem e compilador preliminares. Basicamente, as fases de análise e síntese dessa ferramenta foram refeitas de forma manual, sem o uso de ferramentas próprias para compilação, como *Flex* (2019) e *Bison* (2019) e principalmente sem o uso e amparo do Grafo PON.

Apesar de a LingPON HD 1.0 simplificar a linguagem, reduzindo as entidades para se adequar ao modelo simplificado do PON HD, tal ferramenta é incompatível com o sistema de compilação da LingPON tradicional. Portanto, as aplicações desenvolvidas com uma ferramenta não são compatíveis com a outra, uma vez que as linguagens se diferem inclusive na forma elementar. Além disso, esta nova ferramenta foi desenvolvida sem o uso do Grafo PON, implicando no principal problema que esta tese visa eliminar que é a incompatibilidade entre materializações para o PON. Não somente isso, mas tal ferramenta, assim como outras, normalmente se apresenta de maneira incompleta e, por vezes, inconsistente.

Após constatados todos esses problemas em (KERSCHBAUMER, 2017), subsequentemente, Kerschbaumer (2018) criou uma nova versão de geração de código, especialmente para o PON-HD 1.0, mas desta vez à luz do método MCPON. O código fonte, portanto, é essencialmente baseado na LingPON tradicional e o gerador de código transforma os grafos em códigos em VHDL à luz da tecnologia PON-HD 1.0. Os arquivos VHDL gerados, junto com os componentes do PON-HD 1.0, também descritos em VHDL, podem então ser compilados pelas ferramentas fornecidas pelos fabricantes de hardware.

Na verdade, o arquivo VHDL gerado pela compilação do programa em LingPON-HD 1.0 utiliza instâncias dos componentes do *framework* PON-HD. Cada componente instanciado é parametrizado por meio de sua interface padrão para executar a função desejada, com o número de bits requerido. As interconexões entre os componentes são também realizadas nesse arquivo VHDL. Para fazer essas interconexões são utilizados sinais (“*signal*” em VHDL) do tipo “STD_LOGIC” ou “STD_LOGIC_VECTOR”, conforme a necessidade (KERSCHBAUMER, 2018).

Em suma, a criação desse novo gerador de código seguindo os padrões do MCPON possibilitou que a compilação em PON-HD fosse compatível com o sistema de compilação do PON orientado ao Grafo PON. Diferentemente da ferramenta alternativa criada anteriormente, esta versão se apresenta de maneira completa, seguindo todos os padrões da LingPON 1.2. Nesse âmbito, a compilação para PON-HD se apresenta de forma correta e compatível com as demais materializações do PON. Em tempo, os resultados de performance, de *clock* e uso de espaço em FPGA são corretos e mesmo animadores, conforme detalha Kerschbaumer (2018).

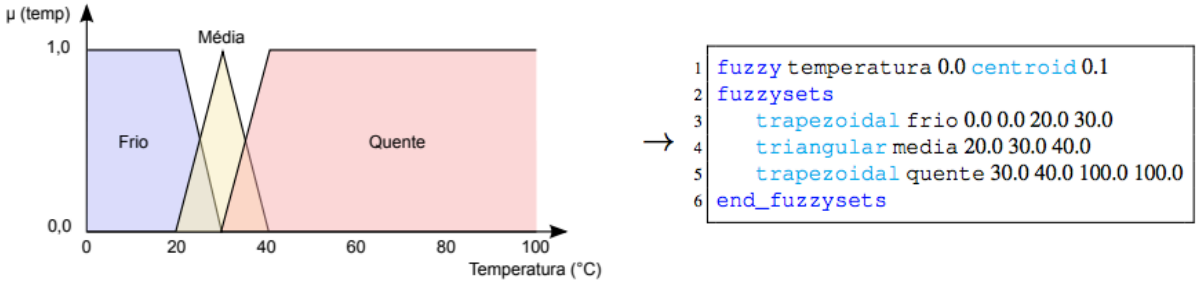
5.5 LINGPON FUZZY

De maneira geral, Melo (2016) adaptou inicialmente o *Framework* PON C++ 2.0 para as particularidades da lógica nebulosa (*fuzzy*), criando uma nova versão deste, uma vez que existem diferenças importantes entre eles. Ainda, Melo (2016) também apresentou um novo dialeto da LingPON, contemplando também as novas características criadas para atender a lógica *fuzzy*. Melo, seguiu as diretrizes do MCPON e construiu sua nova linguagem e compilador (via gerador de código) com base Sistemas de Compilação Preliminar e no Grafo PON preliminar (MELLO *et al*., 2015; MELO, 2016).

A principal mudança na linguagem e compilador associado está relacionada a mudança no estado lógico das entidades do PON. Em suma, as entidades *Premise*, *Condition* e *Rule* apresentam o estado lógico verdadeiro ou false, na teoria original do PON. Para suportar a lógica *fuzzy*, as entidades precisam apresentar uma faixa de valores, no qual está representado seu grau de ativação. O grau de ativação pode assumir qualquer valor real na faixa de 0 a 1. Nesse âmbito, tais entidades iniciarão o processo de notificações, caso o grau de ativação calculado seja diferente do valor atual (MELO, 2016).

Para isso, a linguagem sofreu alterações em sua gramática para suportar as novas estruturas *fuzzy*. A primeira está relacionada a declaração de variáveis linguísticas. Tais variáveis foram modeladas como *Attributes* incluídos em um *FBE*. Para declarar uma variável linguística em um *FBE* na LingPON Fuzzy basta declarar um *Attribute* com o tipo *fuzzy*, conforme ilustra a Figura 71.

Figura 71: Declaração de variável linguística na LingPON *Fuzzy*



Fonte: Melo, 2016

Nesse exemplo, a variável linguística temperatura é declarada por meio de três conjuntos *fuzzy*, um trapézio (frio, entre 0.0 e 30.0 graus, com pertinência máxima entre 0.0 e 20.0 graus), um triângulo (média, entre 20.0 e 40.0 graus, com pertinência máxima em 30.0 graus), e outro trapézio (quente , entre 30.0 e 100.0 graus, com pertinência máxima entre 40.0 e 100.0 graus), como apresentado graficamente na Figura 71. Essa variável linguística é inicializada com ativações para a temperatura 0.0 graus, o método de “defuzificação” a ser utilizado é o de cálculo do centroide (*centroid*), com granularidade 0,1. Internamente, todas as variáveis linguísticas têm seus valores implementados como *Attributes* do tipo *double*, podendo ser utilizadas como tal. A principal diferença é que estes *Attributes* contêm uma lista de conjuntos fuzzy que representam os termos linguísticos da variável (MELO, 2016).

Os *Methods fuzzy*, por sua vez, foram criados para permitir a definição dos consequentes de uma *Rule fuzzy*, isto é, o resultado de uma *Rule* aprovada. Para isso, tais *Methods* são declarados junto aos outros *Methods* tradicionais de um *FBE*. Nesse âmbito, um *Method fuzzy* é declarado utilizando a palavra reservada ***method fuzzy***, conforme exemplificado no Código 49.

Código 49: Exemplo de um *Method fuzzy* na LingPON *Fuzzy*

```
1 method fuzzy temperaturaIsFrio(sensor.temperatura is frio)
```

Fonte: Melo, 2016

Neste exemplo é declarado um *Method fuzzy* que pode ser utilizado no consequente de uma *Rule fuzzy*. Neste caso, a declaração deste pode ser lida da seguinte forma: “A temperatura é frio”.

Para as *Premises fuzzy*, particularmente, foi criado o operador ***is*** que realiza a “fuzzyficação” do valor armazenado na variável linguística para o conjunto *fuzzy* referenciado pela *Premise* conforme apresenta o Código 50.

Código 50: Exemplo de uma *Premise fuzzy* na LingPON *Fuzzy*

```
1 premise sensor.temperatura is frio
```

Fonte: Melo, 2016

As *Premises fuzzy* podem ser utilizadas juntamente com as *Premises* tradicionais em uma *Condition* por meio dos conectivos lógicos definidos na linguagem. No caso das *Instigations fuzzy* foi adicionada uma palavra reservada para identificar que a mesma é *fuzzy* conforme apresenta o Código 51.

Código 51: Exemplo de uma *Instigation fuzzy* na LingPON *Fuzzy*

```
1 instigation fuzzy sensor.temperaturaIsFrio();
```

Fonte: Melo, 2016

Devido as mudanças realizadas na linguagem, Melo criou adaptações tanto na etapa de análise léxica quanto na etapa de análise sintática. Além disso, para o Grafo PON também foram propostas e criadas novas entidades, assim como características adicionais precisaram ser mapeadas, principalmente para mapear as adaptações da linguagem. Ainda, um novo gerador de código foi construído para traduzir as particularidades deste grafo especializado em código PON orientado a lógica *fuzzy*. É importante ressaltar que Melo utilizou o Sistema de Compilação

Preliminar e, por consequência, o método MCPON, como base para a construção de seu compilador *fuzzy*, no qual algumas estruturas do mesmo foram reaproveitadas e readequadas para suportar as adaptações realizadas nesse novo compilador (MELO, 2016).

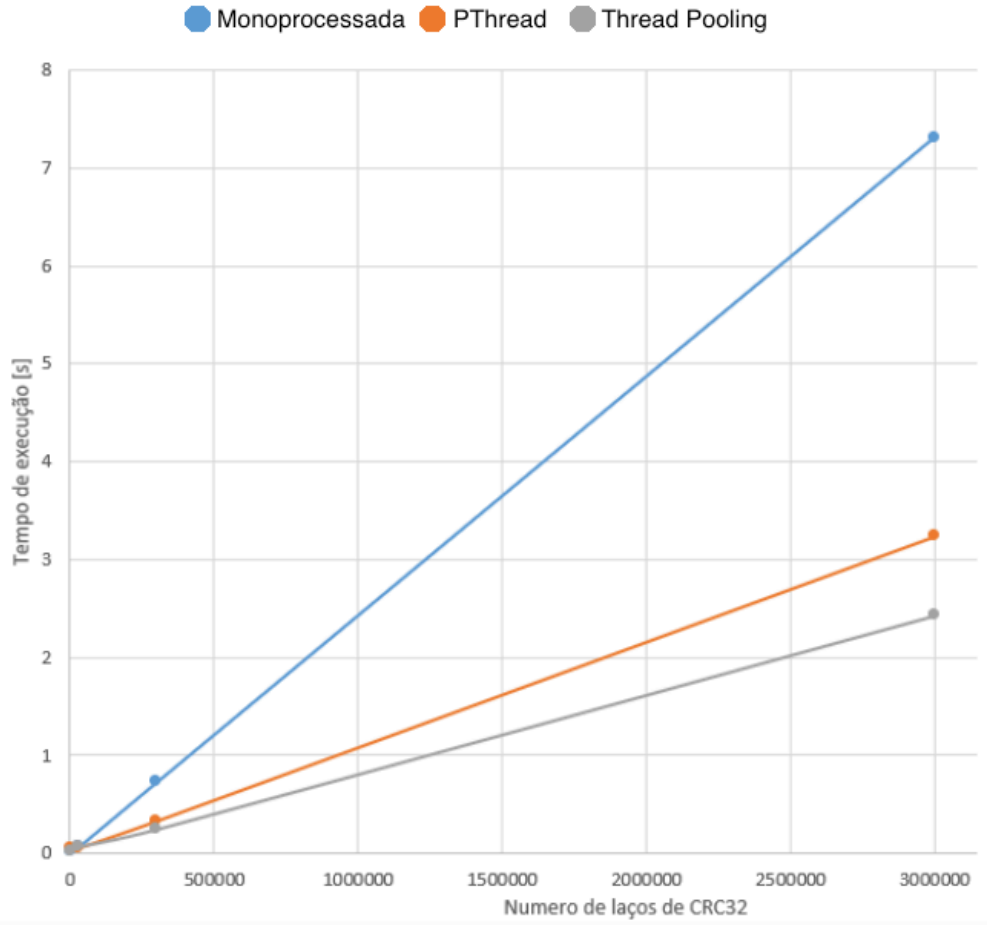
5.6 LINGPON – *ESPAÇO DE NOMES* (NAMESPACE) PARALELA

Martini (2018) adaptou a versão de geração de código *espaço de nomes*, construído inicialmente por Athayde (2016), no âmbito de suportar *multithreading* em computadores x86-64. Na prática, Martini implementou duas abordagens, uma abordagem de paralelização com base na biblioteca tradicional *PThreads* e outra com a técnica *Thread Pooling*.

Basicamente, foram feitas alterações nos *Methods*, particularmente na chamada de cada qual para executarem em *threads*. A escolha de paralelizar apenas as entidades *Method* se deu principalmente porque estas entidades concentram a maior parte do tempo de processamento em programas PON em relação as demais entidades do modelo (MARTINI, 2018).

De modo a testar a eficiência de suas implementações, Martini criou um teste de *benchmark* que calcula repetidas vezes o algoritmo de validação de dados denominado CRC32 para um dado conjunto de bytes. Os cenários incluíram variações do número de execuções do algoritmo com resultados apresentado em escala em segundos, conforme apresenta o gráfico ilustrado na Figura 72.

Figura 72: Comparativo entre versão *Namespaces* mono e multi-processada



Fonte: Martini, 2018

Conforme ilustra a Figura 72, os resultados mostram que ambas as versões paralelizadas apresentam melhor desempenho em comparação com a versão monoprocessada. A versão *Thread Pooling*, porém, apresenta melhor resultado em relação a outra técnica, principalmente por utilizar um *pool* de *threads* em tempo de execução, enquanto para a técnica *PThreads* existe todo um *overhead* de criação e manutenção das *threads* utilizadas durante a execução do programa.

Os resultados de Martini (2018) mostram que as técnicas de *multithreading* podem fornecer melhores resultados de desempenho em tais arquiteturas, porém a falta de implementação de mecanismos de controle que garantam principalmente o determinismo, por hora dificultam o uso destas técnicas em aplicações mais complexas (MARTINI, 2018).

5.7 CONSIDERAÇÕES SOBRE O CAPÍTULO

De maneira geral, a criação de uma linguagem própria para o PON e um sistema de compilação com diversos geradores de código distintos, aqui chamados de Tecnologias LingPON 1.X (*i.e.,* Tecnologia LingPON 1.0, Tecnologia LingPON 1.2, Tecnologia LingPON PON-HD 1.0 e Tecnologia LingPON Fuzzy), representaram um avanço significativo para o paradigma. Em termos de facilidade de programação, tais Tecnologias LingPON 1.X possibilitaram que desenvolvedores pudessem criar programas em PON sem a necessidade de conhecer os conceitos avançados de C/C++, nem as particularidades de cada um dos *frameworks* PON existentes.

Ainda, em termos de facilidade de programação, a codificação em *AssemblyPON* para *NOCA*, bem como a construção de programas baseados em VHDL com base no *Framework* PON-HD 1.0, a partir do advento da Tecnologia LingPON 1.X, já não exigiam mais conhecimento especializado para a utilização de tais tecnologias. Na prática, esta curva de aprendizado foi suavizada por meio da abstração advinda da LingPON e dos compiladores (*i.e.,* gerações de código) pertinentes.

Em termos de desempenho, os modelos de geradores de código para as versões voltadas à plataforma Von Neumann (*e.g., C específico a notificações, C++ estática [a notificações], C++ espaço de nomes [a notificações*] etc.), nomeadamente na Tecnologia LingPON 1.0 e 1.2, possibilitaram criar versões otimizadas baseadas em elementos específicos e, portanto, mais performantes em comparação com o *Framework* PON C++ 2.0. O desempenho superior das versões compiladas estão principalmente associadas a eliminação das estruturas de dados e objetos dinâmicos, os quais se apresentavam como os principais gargalos dos *frameworks*. Assim, o objetivo de alcançar materializações mais eficientes foi atingido e os resultados apresentaram uma evolução considerável nesse âmbito.

Em suma os constituintes da Tecnologia LingPON1.X, nomeadamente as linguagens LingPON, o Sistema de Compilação Preliminar e os geradores de códigos, convergiram para a melhoria do estado da técnica do paradigma PON como um todo. De modo a resumir a efetividade de cada materialização apresentada, a Tabela 7 ilustra as propriedades elementares contempladas nas novas materializações propostas no âmbito da Tecnologia LingPON. Obviamente, esses resultados são fruto da aplicação do MCPON Preliminar.

Tabela 7: Propriedades elementares contempladas nas Tecnologias LingPON

Materialização / Propriedade	Software							Hardware	
	Proto FW 2.0 2013	Proto C a notif. 2013	Proto C++ a n. 2013	1.2 Estática 2015	1.0 Fuzzy 2016	1.2 NS 2016	1.2 NS Thr 2018	1.2 NOCA 2015	1.0 PONHD 1.0 2018
Prog. Alto nível	✓	✓	✓	✓	✓	✓	✓	✓	✓
Paralelismo							✓	✓	✓
Distribuição									
Desempenho		✓		✓		✓	✓	✓	✓

Fonte: Autoria Própria

Conforme apresenta a Tabela 7, todas as materializações advindas da Tecnologia LingPON contemplam a propriedade elementar do PON relacionada a programação em alto nível. Isto já ocorria na Tecnologia LingPON Prototipal que surgiu em 2013, contando com três materializações principais, a versão para *Framework PON* C++ 2.0, a versão C *específica a notificações* (i.e. orientada a funções notificantes) e a versão C++ *específica a notificações* (i.e. orientada a objetos notificantes). Entretanto, no tocante a desempenho/performance, dentre essas materializações, apenas a versão em C a notificações obteve desempenhos comparáveis a implementações de programas similares com base nos princípios do Paradigma Imperativo.

Não obstante, em um segundo momento, os geradores de código na versão estática (a notificações) e, posteriormente, na versão dita *espaço de nomes* (a notificações), foi possível atingir resultados ainda mais performantes do que os da versão em C *especifica a notificações*. Em termos de paralelismo, o gerador de código na versão *espaço de nomes com threads* evidenciou a viabilidade de paralelização dos *Methods*, porém não explorou devidamente o paralelismo efetivo com base nas demais entidades do modelo. Em contrapartida, o gerador de código para o *AssemblyPON* da *NOCA* gera código naturalmente paralelizável pela própria *NOCA*, tal qual sua natureza impele. Aqui a grande vantagem é a programação em alto nível, que foi anexada às propriedades já alcançadas pela própria *NOCA* puramente.

No tocante a Tecnologia LingPON-HD, ela também elimina a necessidade de o desenvolvedor conhecer a linguagem específica de descrição de hardware VHDL e, até mesmo, questões técnicas de hardware, permitindo que a programação aconteça essencialmente em LingPON. A Tecnologia LingPON-HD 1.0 foi baseada no método

MCPON e naturalmente assentada no Sistema de Compilação Preliminar, permitindo a construção de programas em alto nível, apresentando também as propriedades elementares de paralelismo e desempenho efetivo. Com isso, a Tecnologia LingPON-HD 1.0 se tornou a primeira materialização efetiva do PON em um ambiente totalmente paralelo. Assim, à luz da tabela apresentada, faltariam ainda tecnologias que explorem devidamente a distribuição no PON.

Em termos de abrangência de conceitos do PON implementados, a Tecnologia LingPON vem evoluindo ao longo dos anos, por meio das contribuições de pesquisadores e colaboradores do grupo de pesquisa, de modo a contemplar boa parte dos conceitos do paradigma, restando alguns em aberto, entretanto. Para mostrar isso, a Tabela 8 apresenta a plenitude das materializações na Tecnologia LingPON em relação aos conceitos fundamentais do PON.

Tabela 8: Conceitos fundamentais contemplados na Tecnologia LingPON

Materialização / Conceitos	Software							Hardware	
	Proto FW 2.0 2013	Proto C a not. 2013	Proto C++ a n. 2013	1.2 Static 2015	1.0 Fuzzy 2016	1.2 NS 2016	1.2 NS Thr 2018	1.2 NOCA 2015	1.0 PONHD 2018
Reatividade das entidades	✓	✓	✓	✓	✓	✓	✓	✓	✓
Escalonamento de *Rules*									
Estratégias de resolução de conflito									
Compartilhamento de entidades									
Regras de formação				✓		✓	✓	✓	✓
Propriedades reativas dos *Attributes*				✓		✓	✓	✓	✓
Master Rule									
Entidades impertinentes				✓		✓	✓		
FBE Rules				✓		✓	✓		
FBE agregador				✓		✓	✓		

Fonte: Autoria Própria

Em termos de conceitos do PON contemplados nas materializações da Tecnologia LingPON, houve um certo avanço, mas ainda nenhuma delas implementa os conceitos do PON por completo. Nesse âmbito, mesmo que a Tecnologia LingPON-HD 1.0 tenha atingido êxito em demostrar as três principais propriedades elementares

do PON, ela ainda apresenta defeitos como entidades redundantes, devido à falta de suporte ao conceito de *Compartilhamento de Entidades*.

Também falta na Tecnologia LingPON-HD 1.0 suporte a conceitos como *Master Rule*, *FBE Rules* e *FBE agregador*, conceitos estes que poderiam otimizar o modelo mapeado pelo Grafo PON. É importante ressaltar que todos estes conceitos poderiam ser resolvidos a nível de compilação, sem afetar o funcionamento do *framework* PON-HD. Em termos de aplicação dos conceitos no próprio *framework* PON-HD ainda precisaria ser corrigida a questão de *Escalonamento de Rules*, *Estratégias de resolução de conflito* e *Entidades impertinentes*. Com todos estes ajustes, espera-se atingir uma materialização ainda mais efetiva e condizente com os conceitos do paradigma.

Dos conceitos fundamentais, os referentes a compilação poderiam ser resolvidos principalmente com ajustes na linguagem e no processo de compilação, mais precisamente no âmbito do Grafo PON. Certamente, trabalhar na construção de otimizadores, tarefa da Etapa 3 do método MCPON, auxiliaria no amadurecimento de todos os *targets* compilados, uma vez que os problemas estruturais já estariam sido resolvidos no próprio grafo.

Em todo caso, até então, a Tecnologia LingPON toda se constituiu à luz do MCPON Preliminar com sua fidedigna materialização na forma de Sistema de Compilação Preliminar, o qual permitiu a composição da diversidade de geradores de código apresentada ao longo deste capítulo. Em tempo, a Tabela 9 apresenta uma síntese das subetapas do MCPON Preliminar implementadas no âmbito de cada compilador alcançado pela implementação de cada gerador de código no âmbito da Tecnologia LingPON.

Tabela 9: Subetapas do método MCPON contempladas na Tecnologia LingPON.

Materialização / Subetapas	Software							Hardware	
	Proto FW 2.0 2013	Proto C a not. 2013	Proto C++ a n. 2013	1.2 Estática 2015	1.0 Fuzzy 2016	1.2 NS 2016	1.2 NS Thr 2018	1.2 NOCA 2015	1.0 PONHD 2018
Subetapa 1.1 Definição das características da linguagem	✓	✓	✓	✓	✓	✓	✓	✓	✓
Subetapa 1.2 Definição das palavras-chave e analisador léxico	✓	✓	✓	✓	✓	✓	✓	✓	✓
Subetapa 1.3 Definição das regras gramaticais e analisador sintático	✓	✓	✓	✓	✓	✓	✓	✓	✓
Subetapa 2.1 Instanciar as entidade e popular instâncias do Grafo PON	✓	✓	✓	✓	✓	✓	✓	✓	✓
Subetapa 2.2 Construir a integração das análises com o Grafo PON	✓	✓	✓	✓	✓	✓	✓	✓	✓
Subetapa 2.3 Definição das regras semânticas e analisador semântico									
Subetapa 3.1 Criação de otimizadores genéricos independentes de target						✓	✓		
Subetapa 3.2 Criação de otimizadores especializados dependentes de target									
Subetapa 4.1 Iterar instâncias do Grafo PON									
Subetapa 4.2 Construção de geradores de código	✓	✓	✓	✓	✓	✓	✓	✓	✓
Subetapa 5.1 Testes de integridade									
Subetapa 5.2 Compilação de programas completos	✓	✓	✓	✓	✓	✓	✓	✓	✓

Fonte: Autoria Própria

Conforme apresenta a Tabela 9, o MCPON Preliminar foi aplicado apenas de forma parcial na construção da Tecnologia LingPON Preliminar e sistema de compilação associado. Na verdade, naquela época o MCPON não era tratado tão explicitamente como um método para criação de linguagens e compiladores com suas

etapas bem definidas, mas sim como um proto-método na forma de um conjunto de diretrizes e passos a serem seguidos, dos quais muitas delas apenas se apresentavam em forma textual e orientada a exemplos (Ronszcka, 2018).

A falta de um método mais explícito e bem definido dificultou a evolução do ferramental como um todo. Apesar de tais evoluções no âmbito da Tecnologia LingPON, sendo que já havia um Sistema de Compilação associado, muitas delas aconteceram de forma independente e com ramificações paralelas. Em geral, algumas implementações se apresentam de forma não padronizada, desenvolvidas de forma a atender determinada especificação. A título de exemplo, questões que poderiam ser tratadas no Grafo PON, foram implementadas forçadamente a nível de geração de código. Tal medida fez com que determinado *target* atendesse a alguma especificação, mas sem compatibilidade com os demais *targets*.

Além disso, existe uma certa incompatibilidade entre as versões da linguagem, impossibilitando desenvolvedores de compilarem seus códigos em diferentes versões. A mais grave das iniciativas foi a criação de uma LingPON-HD prototipal paralela para o PON-HD 1.0 (KERSCHBAUMER, 2017). Em suma, a criação de uma derivação da linguagem desconectada da Tecnologia LingPON, por um lado possibilitou a criação de facilitadores específicos para a tecnologia PON-HD, porém essa nova linguagem não se tornou compatível com o sistema de compilação do PON. Além disso, as aplicações desenvolvidas com uma versão também não são compatíveis com a outra versão, uma vez que as linguagens se diferem inclusive na forma elementar. Em todo caso, isso foi depois corrigido com a Tecnologia LingPON-HD 1.0 (KERSCHBAUMER, 2018).

Em todo caso, mesmo no tocante a Tecnologia LingPON em si, houve sim um conjunto de dialetos de linguagens e geradores de código que não eram necessariamente compatíveis, conforme já explicado. Assim, essa diversidade de dialetos não comuns dificulta a manutenção e, até mesmo, a aceitação da comunidade para trabalhar com a linguagem e, principalmente, com o paradigma. Nesse âmbito, este trabalho de pesquisa buscou evoluir e amadurecer os conceitos do método MCPON, em particular sobre os conceitos do Grafo PON, de modo a resolver as principais dificuldades encontradas pelos desenvolvedores. Muitas destas questões foram resolvidas e foram propostas na versão dita Efetiva do MCPON, as quais são apresentadas no próximo capítulo.

CAPÍTULO 6

MATERIALIZAÇÃO DO MCPON EFETIVO: SISTEMA DE COMPILAÇÃO PARA LINGUAGEM NOPL

Este capítulo avalia a materialização do método MCPON, agora em sua versão efetiva, denominada de Sistema de Compilação Efetivo. Objetivamente, o Sistema de Compilação Efetivo é a implementação do MCPON Efetivo, por meio de um conjunto de tecnologias, as quais permitem a criação de uma ou mais linguagens, bem como um ou mais compiladores para o PON.

Importante destacar que a proposição do MCPON Efetivo se deu com base no acúmulo de experiências anteriores por meio de implementações em torno do Sistema de Compilação Preliminar. Conforme apresentado no Capítulo 5, esta experimentação permitiu criar inicialmente uma série de linguagens e compiladores para o PON com base no MCPON ainda em sua versão preliminar. Esta experiência acumulada foi de ordem tanto individual do autor da presente tese quanto de ordem coletiva do grupo de pesquisa do PON que colaboraram no desenvolvimento de linguagens e/ou compiladores com base no método proposto.

Em tal grupo de pesquisa, à luz do *know how* acumulado, foram propostas melhorias para o método MCPON em voga e suas materializações, bem como foram instigados alguns questionamentos relevantes sobre a aplicação do método na criação e evolução das linguagens e compiladores particulares ao PON. Ademais, parte considerável deste grupo de pesquisa também se reuniu em um formato estruturado de Grupo Focal (Detalhado na Seção 6.1) para sintetizar o conjunto de considerações e sugestões acumulados durante a primeira fase da pesquisa no tocante ao MCPON Preliminar. O grupo focal em questão contou com a participação de 10 especialistas em PON, além do autor deste trabalho, os quais deliberaram em uma sessão que durou aproximadamente 3 horas. Como resultado, houve importantes vislumbres no estado da arte do método, os quais serão apresentados na Seção 6.1, constituindo um conjunto de demandas.

Subsequentemente na Seção 6.2, a partir das demandas, são apresentados os requisitos para compor a chamada LingPON 2.0 ou NOPL, acrônimo em idioma inglês de *Notification-Oriented Programming Language*. Esta linguagem de programação, em suma, deve ser já bem mais expressiva em relação a sua

predecessora em suas versões, inclusive cobrindo a maior parte dos conceitos atendidos atualmente pelo próprio PON. Naturalmente, a construção da linguagem se dá à luz da teoria geral de linguagens de programação, no âmbito da Etapa 1 do MCPON Efetivo. Justamente, toda a definição e construção da NOPL à luz dos requisitos postos e no MCPON são apresentadas ainda no âmago da Seção 6.2.

Na Seção 6.3 é apresentada primeiramente a implementação do MCPON Efetivo, na forma de um Sistema de Compilação Efetivo que envolve a linguagem NOPL, o Grafo PON Efetivo e todos os demais artefatos necessários. Ainda no seio desta seção em questão são apresentados um conjunto de trabalhos relativos ao Sistema de Compilação Efetivo. Um primeiro trabalho é relativo à expansão natural da NOPL, o que agora é plenamente factível. Outros trabalhos subsequentes são relativos a geradores de códigos para os seguintes ambientes/plataformas: (a) Framework PON C++ 1.0; (b) Framework PON C++ 2.0; (c) Framework PON C++ 3.0 adaptado (Multithread & PON IP); (d) Framework PON Java; (e) Framework PON C#; (f) C++ Namespace com notificações monothread; (g) Assembly NOCA; e (h) Framework PON Erlang/Elixir.

Por fim, na Seção 6.4, são apresentados os resultados da chamada Tecnologia NOPL (ou Tecnologia LingPON 2.0), apresentada na seção 6.3, os quais contemplam a evolução linguística em NOPL, as vantagens do MCPON e Grafo PON implementados na forma de Sistema de Compilação Efetivo em relação ao Preliminar. Derradeiramente, na Seção 6.4, são tecidas as considerações gerais sobre o MCPON no tocante aos avanços por ele proporcionados ao paradigma ao qual se destina.

6.1 GRUPO FOCAL

Para avaliar a proposta desta tese, organizou-se um grupo focal exploratório e confirmatório com especialistas em PON e em LingPON, para avaliar a qualidade e efetividade das tecnologias propostas. Em suma, “grupo focal” é uma técnica de avaliação de artefatos, de natureza qualitativa, que busca o entendimento das considerações e contribuições que um grupo de pessoas teve a partir de uma experiência, ideia ou evento (DRESCH *et al.*, 2014).

Nesse âmbito, os participantes foram convidados com um breve resumo do trabalho que seria realizado e a aplicação desta atividade foi agendada para a data em que o maior número de pessoas poderia estar presente. A condução do grupo

focal foi realizada pelo autor desta tese, o qual participou como moderador da seção. Além disso, foi solicitada e autorizada pelos participantes a gravação em áudio da seção para posterior transcrição e análises, as quais são apresentadas em maiores detalhes no Apêndice E.

Com base nesta atividade, concluiu-se que as considerações apontadas foram bastante pertinentes. Em suma, as principais contribuições foram incorporadas à Tecnologia NOPL como um todo, sendo elas:

a) Modularização do modelo de programação baseada na organização holônica[20] (ou todo-parte) das entidades *FBE*, as quais são estruturadas por meio de escopos locais.
b) *Attributes* internos e externos, permitindo o relacionamento entre os escopos locais (internos) e externos.
c) Possibilidade de "enxertar" códigos da linguagem alvo na definição de *Methods* específicos.
d) Grafo PON com funcionalidades específicas para a navegabilidade nos grafos especializados.
e) Possibilidade de compilar partes de um mesmo programa em plataformas distintas.

Além da validação do próprio método pelos participantes do grupo focal, este encontro apresentou vislumbres importantes para a evolução das tecnologias desenvolvidas. É importante salientar que aqui foi apresentada apenas uma parte das contribuições ou o que foi entendido como consenso. Outras discussões do grupo afloraram nessa atividade e seus desdobramentos também foram considerados em outros pontos da pesquisa. Nesse âmbito, a próxima subseção apresenta o consenso do que fora discutido no âmbito da proposição da NOPL, bem como novos vislumbres subsequentes que levaram a concepção desta nova linguagem de programação para o PON.

[20] O termo programação holônica (derivada do neologismo *holon*) é utilizado no sentido de permitir a relação todo-parte entre instâncias de *FBE* em programas PON. Isto é, uma instância de *FBE* pode ser o "todo" (*holos*, do grego) em seu escopo local, mas apenas "parte" (*on*, do grego, como em partícula *próton, eletrón e nêutron*) dentro de um escopo de outra instância de *FBE* (SIMÃO, 2005).

6.2 LINGUAGEM DE PROGRAMAÇÃO NOPL (LINGPON 2.0)

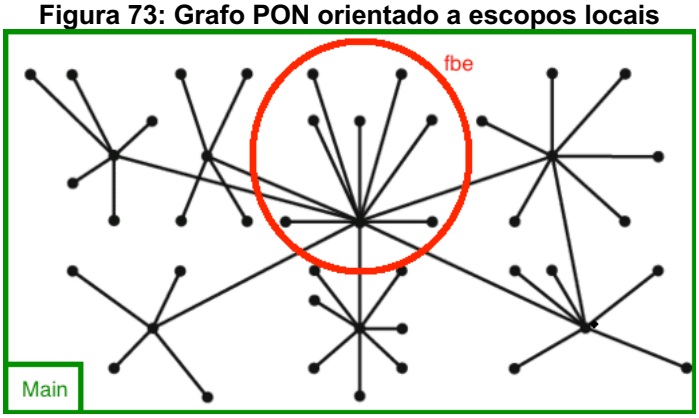
De maneira geral, conforme apresentado no Capítulo 5, as contribuições dos colaboradores que trabalharam na LingPON 1.X, apresentaram propostas de novos conceitos e, principalmente, propostas de melhorias e otimizações para a estrutura da ferramenta como um todo. Entretanto, por falta de um guia de especificações completo e/ou manual de boas práticas, tanto a linguagem quanto o sistema de compilação foram crescendo de forma despadronizada, apresentando dificuldades de integração entre códigos de uma versão para outra e, consequentemente, inibindo a evolução da linguagem e adesão de novos desenvolvedores.

Entretanto, é fato que a linguagem de programação LingPON 1.X, da Tecnologia LingPON Preliminar, possibilitou expressar programas PON em alto nível à luz de sua primeira propriedade elementar. Além disso, associada a um Sistema de Compilação próprio ao paradigma, foi possível demonstrar parcialmente (ou mesmo totalmente, como no caso da Tecnologia PON-HD), as propriedades elementares de desempenho e de desacoplamento para paralelização e/ou distribuição efetivas. Entretanto, alguns pontos ficaram em aberto no tocante a esta Tecnologia e foram discutidos subsequentemente ao seu advento. Muitas das ideias e sugestões coletadas ao longo desse tempo possibilitaram obter uma especificação mais completa e efetiva da linguagem, permitindo não somente evoluir a linguagem em termos de expressividade e completeza, mas também em termos de modelo.

Em suma, a LingPON 1.X apresentava um modelo de programação global, onde as entidades eram definidas e declaradas livremente no código, sem escopos bem definidos. Isso por um lado permitiu demonstrar as propriedades de não-redundância do paradigma, mas não permitiu demonstrar coesão e desacoplamento em termos de modelo. Justamente nesse sentido, as discussões levantadas pelo grupo de pesquisa do PON, incluso no grupo focal apresentado na Seção 6.1, ajudaram a ponderar um modelo de programação holônica, orientada a escopos locais. Tais características, somadas a outras melhorias da expressividade e completeza, auxiliaram na definição de uma nova linguagem para o PON, a qual foi denominada de NOPL, também conhecida como LingPON 2.0. É importante salientar que esta versão da linguagem foi proposta e implementada pelo autor desta tese, com base em partes nas contribuições intelectuais do grupo PON, o que inclui ele mesmo.

Basicamente, na NOPL, um programa deveria ser composto, inicialmente, a partir de um *FBE Main*, o qual seria o elo de ligação (ou o ponto inicial) para a definição de outros *FBEs* e seus relacionamentos. Cada *FBE* nesse caso teria seu próprio escopo interno, onde *Rules* poderiam ser compostas apenas com base nas entidades internas a seu escopo, com a possibilidade de externar *Attributes* para os *FBEs* da camada acima. Na prática, cada *FBE* poderia ser composto por *Attributes, Methods, Rules* e até mesmo por outros *FBEs*, sem limite de níveis/camadas. Isso possibilita a construção modular de programas a nível de modelo. Nesse âmbito, a Figura 73 apresenta um exemplo de Grafo PON descentralizado, à luz desta reestruturação.

Figura 73: Grafo PON orientado a escopos locais



Fonte: Autoria Própria

Conforme ilustra a Figura 73, o Grafo PON orientado a escopos locais, apresenta o *FBE Main* como elo de ligação do programa, composto por *Attributes, Methods, Rules* e outros *FBEs*. Na prática, esse modelo de programação permite dividir o programa em arquivos, onde cada escopo local via *FBE* estaria expresso em seu próprio arquivo de código fonte textual.

É importante ressaltar que esta reestruturação da linguagem não altera o modelo computacional do PON, o qual ainda se apresenta de maneira altamente distribuível. Na verdade, a reestruturação do PON permite alcançar as propriedades de coesão e desacoplamento (tão visada em abordagens com POO) em termos de modelo de programação, justamente pela possibilidade de definição de escopos locais que permitem a construção de programas modularmente.

A especificação dessa nova linguagem tem por objetivo atender de forma universal a criação de aplicações para o PON. Para isso, a linguagem precisa ser completa em termos de abrangência das características e conceitos do paradigma. Nesse âmbito, as subseções subsequentes apresentam a proposta de reorganização

do modelo de programação do PON por meio da linguagem NOPL, contemplando inclusive todos os conceitos vislumbrados na Seção 3.6, visando desta forma, a plenitude da linguagem.

6.2.1 Organização holônica das entidades *FBE*

Em linhas gerais, a organização da linguagem NOPL segue um modelo de programação baseado na organização holônica das entidades do PON, as quais são estruturadas por meio de escopos locais. É importante salientar que essa organização holônica foi uma contribuição elencada pelo grupo de pesquisa do PON, mais particularmente relacionada a conclusão *a)* da Seção 6.1.

Tal modelo se diferencia da versão precedente da linguagem, a qual é baseada apenas em um único escopo global. Essa reestruturação do modelo de programação permite maior organização dos elementos de um programa, proporcionando mais facilidade de manutenção de código. Nesse âmbito, o Código 52 apresenta um exemplo de programa PON por meio da organização holônica dos *FBEs* na linguagem NOPL.

Código 52: Organização holônica dos *FBEs* na linguagem NOPL

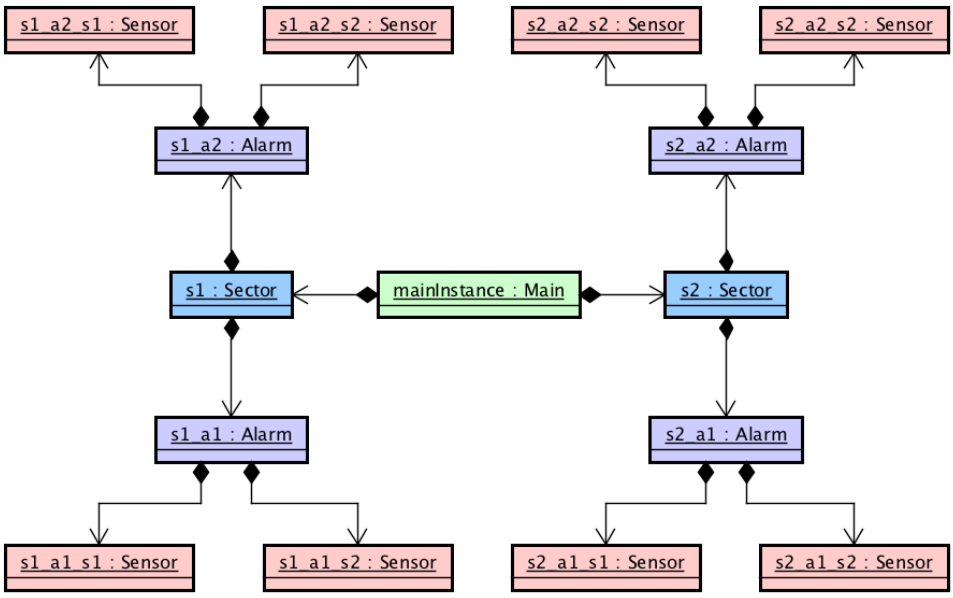
```
1  //arquivo sensor.pon
2  fbe Sensor
3  end_fbe
4
5  //arquivo alarm.pon
6  fbe Alarm
7    Sensor s1
8    Sensor s2
9  end_fbe
10
11 //arquivo sector.pon
12 fbe Sector
13   Alarm a1
14   Alarm a2
15 end_fbe
16
17 //arquivo main.pon
18 fbe Main
19   Sector s1
20   Sector s2
21 end_fbe
```

Fonte: Autoria Própria

Conforme apresenta o Código 52, o programa é organizado por meio da definição de *FBEs*, cada qual em um arquivo próprio e internamente por meio da

composição de instâncias de *FBEs*, definindo uma relação holônica (todo-parte) entre elas. Conforme ilustra o programa exemplo, o *FBE* principal é composto por duas instâncias do *FBE Sector* (*s1* e *s2*), onde cada qual é composto por duas instâncias do *FBE Alarm* (*a1* e *a2*), e por fim, cada qual por instâncias do *FBE Sensor* (*s1* e *s2*), conforme ilustra o diagrama de objetos em UML modelado na Figura 74.

Figura 74: Relação holônica entre instâncias de *FBE*



Fonte: Autoria Própria via Diagrama de objetos em UML

6.2.2 Encapsulamento dos *FBEs*

Conforme contribuição elencada pelo grupo de pesquisa do PON, mais particularmente relacionada a conclusão *b)* da Seção 6.1, o encapsulamento dos *FBEs* corrobora com a organização holônica das entidades. Para tal, a NOPL permite o encapsulamento de instâncias de *Attributes* e instâncias de *FBEs* (agregadas), assim como definições de *Methods* particulares a cada *FBE*. Nesse ínterim, também é possível externar (ou tornar públicas) cada uma destas. De modo a apresentar estas particularidades no âmbito da linguagem NOPL, o Código 53 apresenta exemplos de encapsulamentos.

Código 53: Exemplos de encapsulamentos na linguagem NOPL

```
1  fbe Main {
2
3    private boolean atStatus = false;
4
5    private method mtChange() {
6        this.atStatus = true;
7    }
8
9    rule rlChange(this.atStatus == true) {
10     this.mtChange();
11   }
12
13   main {
14     this.atStatus = true;
15   }
16
17 }
```

Fonte: Autoria Própria

Conforme apresenta o Código 53, cada entidade *FBE* pode apresentar em sua definição objetos privados, os quais são visíveis apenas internamente ao escopo desta, assim como objetos públicos, os quais são visíveis externamente ao escopo local de tal *FBE*. A linguagem NOPL, portanto, permite internalizar um modelo de entidade, por meio de instâncias de *Attributes* e *FBEs*, bem como por meio de definições de *Methods.*

Ademais, o conceito de encapsulamento é bastante comum em diversas linguagens de paradigmas usuais, as quais geralmente são linguagem-alvo dos compiladores (via geradores de código) resultantes do Sistema de Compilação do PON. Nesse âmbito, o suporte ao encapsulamento na NOPL, auxilia na tradução do código fonte para as linguagens alvo definidas. Além disso, essa reorganização holônica com possibilidade de externar apenas as entidades pertinentes, auxilia na garantia de coesão e desacoplamento dos elementos de um programa PON.

6.2.3 Redefinição dos *Methods*

De maneira geral, a estrutura de definição de um *Method* não seguia uma certa padronização em relação aos demais blocos e instruções da LingPON, muito menos em relação a outras linguagens tradicionais. Basicamente, a atribuição a ser realizada pelo *Method* era informada por meio de parênteses, o que gerava certa confusão na sua utilização, conforme apresentado no Código 19 na Seção 5.1.1.

Nesse âmbito, de modo a evitar confusões sobre como um *Method* deveria ser definido e utilizado, a NOPL apresenta uma nova forma para a construção de *Methods*. Essa definição foi proposta pelo autor desta tese em sua qualificação em (RONSZCKA, 2018), e adequada as evoluções da linguagem, sendo inclusive corroborada pelo grupo de pesquisa do PON. Nesse âmbito, o primeiro caso que representa a atribuição de um valor qualquer a um dado *Attribute* é apresentado em forma de exemplo no Código 54.

Código 54: Definição de um *Method* de atribuição na linguagem NOPL

```
1  private method mtRingTheSiren
2    assignment
3      atTimer = 60
4    end_assignment
5  end_method
```

Fonte: Autoria Própria

Conforme apresentado no Código 54, a definição de um *Method* é realizada por meio das palavras reservadas ***method*** e ***end_method***. Ainda, as palavras reservadas ***assignment*** e ***end_assignment*** definem que este *Method* será responsável pela atribuição de um dado valor para um *Attribute* específico. Além disso, é importante observar que o *Method* só pode ser composto por uma única instrução de atribuição, garantindo assim, o cumprimento de um dos princípios de programação denominado de *Single Responsibility Principle*[21], o qual visa preservar a coesão de um programa.

Ademais, um *Method* também apresenta a possibilidade de atribuir o resultado de uma expressão aritmética à um dado *Attribute*. Essa característica é apresentada no exemplo do Código 55.

Código 55: Definição de um *Method* com expressão aritmética na NOPL

```
1  private method mtCountOne
2    arithmetic
3      atCounter = atCounter + 1
4    end_arithmetic
5  end_method
```

Fonte: Autoria Própria

[21] *Single Responsibility Principle* – é um princípio de programação que afirma que cada módulo de software (*e.g.* classe, método) deve ter responsabilidade sobre uma única parte da funcionalidade fornecida pelo *software*, e essa responsabilidade deve ser totalmente encapsulada por esse módulo (RONSZCKA, 2012).

Conforme apresenta o Código 55, as palavras reservadas ***arithmetic*** e ***end_arithmetic*** definem que este *Method* será responsável pela execução da expressão aritmética seguida da atribuição do resultado da expressão para um *Attribute* específico.

Apesar das expressões aritméticas serem bastante utilizadas na programação para as mais diversas finalidades, elas por si só não são suficientes para contemplar todas as possibilidades no âmbito da programação. Além disso, assim como em outras linguagens de programação tradicionais, existe a necessidade de criar *Methods* genéricos que possam atuar sobre parâmetros dinâmicos e, ainda, retornar um valor após a execução do mesmo. Nesse âmbito, a NOPL apresenta uma nova forma de definir *Methods* com parâmetros dinâmicos e o retorno explítico do valor de sua execução. Para isso, o Código 56 apresenta a definição de um *Method* com parâmetros e retorno.

Código 56: Definição de *Method* com parâmetros e retorno na linguagem NOPL

```
1  private method mtCalcHypotenuse
2    params
3      double x
4      double y
5    end_params
6    arithmetic
7      sqrt((x * x) + (y * y))
8    end_arithmetic
9    return double
10 end_method
```

Fonte: Autoria Própria

Conforme apresenta o Código 56, o exemplo em questão implementa o cálculo da Hipotenusa, conforme teorema de Pitágoras, o qual define que o quadrado da hipotenusa é igual à soma dos quadrados dos catetos. Nesse exemplo, em particular, é apresentado a definição de parâmetros em um *Method*. Para isso é necessário utilizar as palavras reservadas ***params*** e ***end_params*** bem como definir o tipo e o nome dos parâmetros desejados, conforme linhas 2 a 5. Ademais, a linha 7 apresenta o respectivo uso destes parâmetros em uma expressão aritmética composta, bem como o retorno do resultado da equação, conforme linha 9.

É pertinente salientar que a composição da execução de um *Method* pode tanto fazer uso de parâmetros genéricos quanto fazer uso de *Attributes* do *FBE* em questão. Ademais, o *Method* pode atribuir o resultado da expressão em um *Attribute*

do *FBE*, ao invés de retornar o valor da expressão para o elemento que chamou o *Method*.

Ainda, após as definições dos *Methods*, eles podem ser chamados por meio da entidade auxiliar *Call*, a qual é definida no âmbito da entidade *Instigations.* É importante ressaltar que as entidades *Method* precisam estar visíveis no escopo da *FBE* para a qual estas chamadas são realizadas. Nesse caso, cada chamada de *Method* é realizada a partir de uma instância de um *FBE* e pode conter um conjunto de parâmetros em sua essência, assim como também permite definir um *Attribute* para '*setar*' o valor do retorno deste *Method*, caso tenha o mesmo tenha um valor de retorno. Nesse sentido, o Código 57 exemplifica a utilização de chamadas de *Method* na NOPL.

Código 57: Chamadas de *Methods* na NOPL

```
1  instigation
2    call sirenA.mtRingTheSiren
3    call this.mtCalcHypotenuse
4      params
5        rectangle.atValueA
6        rectangle.atValueB
8      end_params
9      set rectangle.atValueC
10   end_call
11 end_instigation
```

Fonte: Autoria Própria

Conforme apresenta o Código 57, o primeiro *Call* de *Method* é realizado basicamente a partir de uma instância de um *FBE* visível no escopo da *Rule/FBE* em questão. Tal chamada é anunciada pela palavra reservada ***call***. O segundo *Call* de *Method*, por sua vez, apresenta algumas novidades, como a definição de um bloco representado pelas palavras reservadas ***call*** e ***end_call***, bem como o uso das palavras reservadas ***params*** e ***end_params*** para a passagem de parâmetros. Além disso, também apresenta a palavra reservada ***set***, a qual é utilizada para mapear o valor de retorno da execução desse *Method* a um dado *Attribute* do programa.

É importante ressaltar que, diferentemente das versões anteriores da linguagem, o uso de parênteses foi eliminado tanto da definição de um *Method* quanto de sua utilização, tornando a linguagem mais padronizada em relação a seu conjunto de blocos e formato de definições e declarações.

6.2.4 *Methods* integráveis com a linguagem do *target*

Desde a versão prototipal da LingPON, existiu o interesse em integrar o código alvo gerado à luz do PON (*i.e.,* código orientado a notificações) na linguagem *target* à códigos de programas legados nesta mesma linguagem. Isso já era feito com todas as versões do *Framework* PON C++, no qual eram criados *Methods* específicos para invocar funções de bibliotecas de terceiros ou, até mesmo, métodos de classes orientadas a objeto.

Na prática, nos *frameworks*, a interligação entre os dois paradigmas acontece por meio da entrada de dados, com um *Attribute* tendo seu valor definido pelo código legado ou, por meio da execução de alguma rotina, com o próprio PON chamando funções/métodos de módulos importados. Essa integração acontece totalmente de forma manual, onde o programador precisa fazer alterações no código PON e inclusive no código legado para conectar suas classes com o mecanismo de inferência do PON.

Nas primeiras versões da LingPON, essa integração era ainda mais custosa, pois não existiam mecanismos próprios para a inserção de código legado na linguagem. Dessa forma, a cada compilação do programa, era necessário reorganizar o código gerado para integrar com o código legado.

Ademais, no âmbito da Tecnlogia LingPON 1.2, com o surgimento da LingPON com geração de código para versão C++ Estática (Seção 5.3.2.1), foram criados mecanismos improvisados para a interpretação de expressões aritméticas, variáveis locais e atribuições múltiplas, traduzindo esse novo código para a linguagem alvo. Essa abordagem, além de dificultar o entendimento de um programa escrito em LingPON, recriou elementos da programação imperativa que não fazem parte da essência do PON (*i.e.* variáveis passivas e locais).

Além disso, amarrou a geração de código para versão C++ Estática para um único alvo, compilado para C++ com elementos estáticos, tornando assim a linguagem menos universal e não compatível com as demais versões. Outrossim, como a versão Estática era totalmente orientada a variáveis estáticas, existiu uma dificuldade para a integração do código gerado pelo compilador com o código legado orientado a objetos, uma vez que atributos estáticos só podem ser acessados por métodos estáticos.

Sendo assim, surgiu a necessidade de criar um novo alvo, orientado a variáveis não estáticas, mas que ainda assim, preservasse a eficiência advinda das

otimizações implementadas nessa versão. Para isso, no âmbito da Tecnologia LingPON 1.2, foi criada a versão do gerador de código denominado Espaço de Nomes (*Namespaces*) (cf. Seção 5.3.2.2). Junto com ela foi proposto um mecanismo de integração preliminar que demonstrou ser possível utilizar esse estilo de geração de código para a integração com código legado.

Apesar do mecanismo de integração preliminar demonstrar sua viabilidade, a especificação definida para a LingPON não seguia os padrões da linguagem. Dessa forma, esse mecanismo foi redefinido na NOPL com base nas sugestões elencadas pelo grupo de pesquisa do PON, mais particularmente relacionada a conclusão *c)* da Seção 6.1. Nesse âmbito, um exemplo desta implementação é apresentado no Código 58.

Código 58: *Methods* para integração com códigos legados na linguagem NOPL

```
1  private method mtSaveDeviceData
2    params
3      string data
4    end_params
5    code C++
6      DatabaseCPP *db = new DatabaseCPP();
7      db->insertDeviceData(data);
8    end_code
9    code Java
10     DatabaseJava db = new DatabaseJava();
11     db.insertDeviceData(data);
12   end_code
13 end_method
```

Fonte: Autoria Própria

Conforme apresenta o Código 58, o *Method* integrável para execução de chamadas de funções/métodos na linguagem alvo apresenta como novidade as palavras reservadas ***code*** e ***end_code***, seguidas do identificador da linguagem alvo. É importante ressaltar que o compilador, em tempo de compilação, deveria vincular o *target* com o parâmetro especificado, “enxertando” o código entre o bloco *code* na íntegra no código-alvo gerado. Ademais, nesse bloco é possível criar e utilizar (ainda que nem sempre necessariamente aconselhável) todos os elementos da programação da linguagem alvo, como estruturas de decisão e repetição, variáveis locais, alocação dinâmica de memória etc.

É importante observar que, nesse novo modelo, o compilador vai integrar os parâmetros do *Method* com o uso do código legado e, inclusive, conectar o retorno do código legado com o retorno do *Method*, possibilitando a entrada de dados no

mecanismo de inferência do PON por meio da execução de métodos/funções implementados no código legado, eliminando a dificuldade de integração entre códigos compilados e códigos legados.

6.2.5 Inclusão de código externo na linguagem do *target*

Relacionado com a subseção anterior, durante a programação para o PON, principalmente na criação de programas mais elaborados, há a necessidade de incluir bibliotecas de terceiros e definir variáveis ou objetos globais, dependendo do caso. Na prática, esta seção define um novo bloco dentro da linguagem NOPL para definições sem restrições, permitindo qualquer definição da linguagem *target* dentro deste bloco. A principal diferença em relação ao mecanismo anterior é que os *Methods* executam funções específicas conforme os *calls* do programa PON, enquanto este novo bloco permite qualquer outra interação com a linguagem alvo, como declaração e inicializações de variáveis e definições de constantes. Ademais, tal integração não se limitaria a apenas integrar blocos de código legados, mas inclusive, facilitaria a integração com bibliotecas de terceiros, no âmbito de criar aplicações com acesso a banco de dados, drivers de hardware ou comunicação via rede com servidores e afins.

Em suma, esse bloco complementa a chamada de *Methods* apresentada anteriormente, no sentido de preparar o ambiente de execução de acordo com as especificações definidas. Nesse sentido, um bloco opcional é proposto para inclusão de códigos na linguagem alvo, conforme apresenta o exemplo no Código 59.

Código 59: Bloco opcional para códigos externos na linguagem NOPL

```
1  external C++
2
3    #include <iostream>
4    using namespace std;
5
6    #include "SQL.h"
7
8    #include "caminho/arquivo/MinhaClasse.h"
9
10   static const float PI =  3.14159265359;
11
12 end_external
```

Fonte: Autoria Própria

Conforme apresenta o Código 59, o bloco para inclusão de códigos externos é anunciado pelas palavras reservadas ***external*** e ***end_external***, seguidas do

identificador da linguagem alvo. É importante ressaltar que o compilador, em tempo de compilação, deveria vincular o *target* com o parâmetro especificado, “enxertando” o código na íntegra no código-alvo gerado. Nesse bloco, é possível incluir bibliotecas padrão da linguagem C/C++ (linha 3), definições de *namespaces* (linha 4), inclusões de classes de terceiros (linha 6), inclusões de classes próprias (linha 8), definições de constantes etc. Ademais, este bloco é tido como opcional, podendo não ser utilizado na criação de programas na linguagem NOPL.

6.2.6 Parâmetros de execução sequencial ou paralela

De maneira geral, a LingPON, até então, nunca tratou de forma clara a diferença entre a execução de *Methods* de maneira sequencial ou paralela. Isso normalmente ficou a cargo dos compiladores que, ou transformavam tudo em execução sequencial, no caso da maioria dos geradores de código para linguagens alvo baseadas na arquitetura Von Neumann, ou transformavam tudo em execução paralela, no caso da compilação para VHDL. Entretanto, existem casos em que mesmo que o ambiente seja totalmente paralelizável, a execução sequencial de alguns métodos precisa ser mantida, para garantir a sequencialidade de execução de um algoritmo, por exemplo.

Essa definição foi proposta pelo autor desta tese em sua qualificação (2018) e adequada as evoluções da linguagem, sendo inclusive corroborada pelo grupo de pesquisa do PON. Nesse sentido, um parâmetro foi criado para definir se um conjunto de *Instigations* deve executar de forma sequencial ou paralela no âmbito de uma *Action* e se um conjunto de *Methods* deve executar de forma sequencial ou paralela no âmbito de uma *Instigation*. Esse parâmetro deve ser informado logo após as palavras reservadas ***action*** ou ***instigation***. Até o momento, tal parâmetro é configurado como (a) ***sequential*** para a execução sequencial de um conjunto de *Methods* e (b) ***parallel*** para a execução paralela do respectivo conjunto. De modo a exemplificar o uso de tal parâmetro, o Código 60 apresenta um exemplo de uma *Rule*.

Código 60: Definição de parâmetros de execução sequencial ou paralela

```
1  rule rlFireAlarm
2    ...
3    action sequential
4      instigation parallel
5        call mtRingSiren1
6        call mtRingSiren2
7      end_instigation
8      instigation
9        call mtNotifyAllUsers
10     end_instigation
11   end_action
12 end_rule
```

Fonte: Autoria Própria

Conforme apresenta o Código 60, a *Rule rlFireAlarm*, responsável pelo disparo de um alarme hipotético, apresenta a execução da *Action* de maneira sequencial (linha 3), onde a *Instigation* da linha 4 precisa executar por completo para, só então, executar a *Instigation* da linha 8. É importante observar que a primeira *Instigation* é definida para executar de forma paralela. Nesse caso, tanto o *Method* da linha 5 quanto o da linha 6 podem executar de forma paralela.

Outrossim, no âmbito de uma *Instigation*, independente de ser sequencial ou paralela, todos os *Methods* precisam retornar suas chamadas para que a *Instigation* seja finalizada. Sendo assim, caso uma *Action* seja definida para executar de forma sequencial, cada *Instigation* fica condicionada ao término da *Instigation* anterior para iniciar suas chamadas de *Métodos*.

6.2.7 Redefinição do mecanismo de propriedades reativas dos *Attributes*

De maneira geral, em dado momento, foi embutido na linguagem um mecanismo improvisado para tratar os conceitos de renotificação e de não-notificação, conforme explicado na Seção 5.3.1.4. Entretanto, o modelo não havia sido documentado de forma apropriada e também não se encaixava nos padrões da linguagem (*i.e.* com clareza vocabular e orientada a blocos bem organizados). Nesse âmbito, o conceito é redefinido e apresentado aqui nesse trabalho de uma forma mais clara e definitiva.

Na prática, quando um *Method* é instigado, ele normalmente faz a alteração de um *Attribute*, seja no corpo da definição do *Method* ou no retorno de seu valor. Em ambos os casos, o comportamento padrão da inferência do paradigma é continuar a cadeia de notificações caso o valor do *Attribute* tenha sido alterado.

Entretanto, para casos onde se deseja que a cadeia de notificações seja continuada mesmo sem a alteração efetiva do valor do respectivo *Attribute*, o parâmetro ***renotify*** deve ser utilizado. Ainda, existem casos em que a propagação da notificação deve ser contida, para isso o parâmetro ***nonotify*** deve ser utilizado. Nesse âmbito, o Código 61 apresenta um exemplo da utilização de tais parâmetros na linguagem NOPL.

Código 61: Exemplo dos parâmetros *renotify* e *nonotify* na linguagem NOPL

```
1  call renotify mtResetTimer
2  call nonotify mtDecrementTimer
```

Fonte: Autoria Própria

Conforme apresenta o Código 61, na linha 1, a utilização do parâmetro ***renotify***, em um exemplo hipotético de *Method* para resetar um temporizador de um sistema de alarme, onde mesmo que o estado não tenha sofrido alterações (*i.e.* múltiplos *resets*), é desejável que ocorra a propagação das notificações para a realização das demais checagens e eventual continuação do fluxo de execução do programa. Na linha 2, por sua vez, é apresentada a utilização do parâmetro ***nonotify***. Tal parâmetro deve ser utilizado pelo desenvolver quando este não desejar que uma alteração no estado de um determinado *Attribute* propague notificações para as *Premises* associadas.

6.2.8 Dependência entre *Rules* – *Master Rule*

Assim como apresentado no *Framework* PON C++ 2.0 (conforme Seção 3.6.5), existem casos que um conjunto considerável de *Rules* dependeria de *Premises* semelhantes para suas aprovações/execuções. A criação de entidades *Premise* únicas (com o mesmo teste lógico causal) e seu respectivo compartilhamento seria a alternativa mais apropriada nesse cenário, evitando dessa forma a presença de entidades redundantes. Entretanto, existem outros casos em que tais *Rules* dependeriam de duas ou mais *Premises* compartilhadas, as quais notificariam todas as entidades interessadas em mudanças ocorridas em seu estado.

Nesse sentido, notificações desnecessárias poderiam ser evitadas caso as *Premises* comuns a todas as *Rules* notificassem apenas uma *Rule*, ao invés de todas elas. Assim, no momento que todas as *Premises* dessa *Rule* única apresentassem

estado verdadeiro, a ponto de aprovar a execução dessa, ela notificaria as demais *Rules* interessadas. Nesse âmbito, de modo a evitar tal sobrecarga de processamento e mesmo dificuldades de implementação (*i.e.,* compartilhamento de entidades semelhantes e passividade a erros), a funcionalidade de *Rules* dependentes permite criar uma dependência entre *Rules* no PON, onde uma ou mais *Rules* dependeriam da execução de uma determinada *Rule* para, só então, executarem.

Nesse caso, tais *Premises* fariam parte de uma única *Rule*, a qual faria o papel de *Rule* mestre. As demais *Rules* fariam um vínculo com essa *Rule*, de tal forma que dependeriam de sua aprovação e respectiva notificação para só então executarem. Essa dependência traria benefícios em questões de desempenho e facilidades na composição de aplicações. De modo a apresentar como esse conceito é implementado na linguagem NOPL, o Código 62 apresenta um exemplo de dependência entre *Rules*.

Código 62: Dependência entre *Rules* – *Master Rule* na linguagem NOPL

```
1  rule rlFireAlarm
2    depends
3      rlSectorAFired
4      or
5      rlSectorBFired
6      or
7      rlSectorCFired
8    end_depends
9    condition
10     . . .
11   end_condition
12   action
13     . . .
14   end_action
15 end_rule
```

Fonte: Autoria Própria

Conforme apresenta o Código 62, as palavras reservadas ***depends*** e ***end_depends*** foram introduzidas para a definição de dependência entre *Rules*. Ademais, é possível criar uma dependência múltipla com diversas *Rules*, seja de forma disjuntiva (*or*), onde qualquer uma das *Rules* aprovadas tornam a *Rule* elegível para aprovação, ou de forma conjuntiva (*and*), onde a *Rule* só se torna elegível para aprovação quando todas as demais *Rules* estiverem aprovadas.

6.2.9 Exemplo completo do uso da nova versão da linguagem

De modo a apresentar um exemplo mais completo do uso da linguagem NOPL, o Código 63 apresenta um exemplo do programa Sensores, reconstruído de acordo com a maioria das modificações e novos conceitos apresentados na Seção 6.2.1.

Código 63: Reconstrução do programa Sensores baseado na linguagem NOPL

```
1  //arquivo sensor.pon
2  fbe Sensor
3
4      public boolean atState = false
5
6  end_fbe
7
8  //arquivo siren.pon
9  fbe Siren
10
11     public integer atTime = 0
12
13 end_fbe
14
15 //arquivo alarm.pon
16 fbe Alarm
17
18     public boolean atStatus = false
19
20 end_fbe
21
22 //arquivo sector.pon
23 fbe Sector
24
25     private Boolean atIntruderDetected = false
26
27     private Alarm alarmA
28     private Alarm alarmB
29
30     private Siren sirenA1
31     private Siren sirenA2
32     private Siren sirenB1
33
34     private Sensor sensorA1
35     private Sensor sensorA2
36     private Sensor sensorB1
37
38     private method mtNotifyInvasion
39         assignment
40             this.atIntruderDetected = true
41         end_assignment
42     end_method
43
44     rule rlFireAlarmA
45         condition
46             subcondition
47                 premise prSectorInPeaceA
```

```
48                      this.atIntruderDetected == false
49                  end_premise
50                  and
51                  premise prAlarmAOn
52                      alarmA.atStatus == true
53                  end_premise
54              end_subcondition
55              and
56              subcondition
57                  premise prSensorA1State
58                      sensorA1.atState == true
59                  end_premise
60                  or
61                  premise prSensorA2State
62                      sensorA2.atState == true
63                  end_premise
64              end_subcondition
65          end_condition
66          action sequential
67              instigation parallel
68                  call this.mtNotifyInvasion
69              end_instigation
70          end_action
71      end_rule
72
73      rule rlFireAlarmB
74          condition
75              premise prSectorInPeaceB
76                  this.atIntruderDetected == false
77              end_premise
78              and
79              premise prAlarmBOn
80                  alarmB.atStatus == true
81              end_premise
82              and
83              premise prSensorB1State
84                  sensorB1.atState == true
85              end_premise
86          end_condition
87          action sequential
88              instigation sequential
89                  call this.mtNotifyInvasion
90              end_instigation
91          end_action
92      end_rule
93
94      properties
95          strategy PRIORITY
96      end_properties
97
98  end_fbe
99
100 //arquivo main.pon
101 fbe Main
102
103     external NAMESPACES
104         #include "SMSSender.h"
105         #include <iostream>
106         using namespace std;
107     end_external
108
```

```
109     private Sector sectorA
110     private Sector sectorB
111
112     private method mtSendSms
113         params
114             String cellphone
115         end_params
116         code NAMESPACES
117             SMSSender *sender = new SMSSender();
118             sender->send(cellphone);
119         end_code
120     end_method
121
122     rule rlInvasionDetection
123         condition
124             premise prSectorAInvaded
125                 sectorA.atIntruderDetected == true
126             end_premise
127             or
128             premise prSectorBInvaded
129                 sectorB.atIntruderDetected == true
130             end_premise
131         end_condition
132         action parallel
133             instigation parallel
134                 call this.mtSendSms
135                   params
136                     "41-999999999"
137                   end_params
138                 end_call
139           end_instigation
140         end_action
141     end_rule
142
143     properties
144         strategy PRIORITY
145     end_properties
146
147     main
148       sectorA.atIntruderDetected = false
149       sectorB.atIntruderDetected = false
150     end_main
151
152 end_fbe
```

Fonte: Autoria Própria

No Código 63, mais especificamente, nas linhas 2 a 6, é apresentada a definição de um FBE *Sensor*, assim como as linhas 9 a 13 apresentam a definição do *FBE Siren* e as linhas 16 a 20 apresentam o *FBE Alarm*. É importante salientar que cada um destes *FBEs* é definido em arquivos distintos, cada qual representando seu próprio escopo local, facilitando a organização holônica, conforme detalhado anteriormente.

Ademais, na linha 23 é iniciado um *FBE* mais completo, a declaração do *FBE Sector*. Basicamente, ele apresenta um *Attribute* privado (linha 23) e um conjunto de

instâncias dos três *FBEs* declarados anteriormente (linhas 27 a 36). Nas linhas 38 a 42 é definido um *Method* de atribuição, o qual basicamente 'seta' o valor do *Atribute* privado deste *FBE* para *true*. Este *FBE* define duas *Rules* principais, tanto para o monitoramento do *sectorA* quanto para o monitoramento do *sectorB*.

É importante salientar que este programa poderia ser reescrito de forma a contemplar a *Rule* de monitoramento internamente ao *FBE Alarm*, desta forma, não seria necessário replicar as instâncias e *Rules* para cada qual, mantendo apenas uma única *Rule* no escopo local de tal *FBE*. Nesse âmbito, seria necessário externar o estado de um *Attribute* de *Alarm* para então compor uma *Rule* mais enxuta na camada acima (*Sector*). Essa relação hierárquica entre os *FBEs* permite a composição do programa de maneira coesa, facilitando a legibilidade e codificação do programa como um todo.

Por fim, a definição do *FBE Main* (linha 101) apresenta o bloco *external* (linhas 103 a 107) que basicamente incluem as bibliotecas necessárias para a compilação e execução do *Method* com código especializado no *target* (linhas 112 e 120). As linhas 134 a 138 fazem a chamada de tal *Method,* passando como parâmetro um valor estático. Ademais, nas linhas 143 a 145 é definido o bloco de propriedades que basicamente configura o *FBE* para atuar de acordo com o especificado. No âmbito do próprio do *FBE Main*, como concentrador dos principais *FBEs* do "universo", está a própria inicialização do sistema esboçada no bloco main, conforme linhas 147 a 160.

De maneira geral, a proposta dessa nova versão da linguagem apresenta uma escrita fluida e padronizada, eliminando as características implementadas de modo improvisado e, principalmente, apresenta um formato universal para a composição de programas em PON, contemplando inclusive as características e conceitos do paradigma em sua totalidade. Ademais, as melhorias da linguagem são pertinentes para reforçar a propriedade elementar do PON que visa facilitar o desenvolvimento em alto nível com base em uma estruturação bem organizada dos elementos do modelo do paradigma. Além disso, tal linguagem foi desenvolvida para ser extensível por outros desenvolvedores, como será relatado na próxima seção.

6.3 IMPLEMENTAÇÃO DO MÉTODO MCPON EFETIVO

Esta seção apresenta a implementação do método MCPON, em sua versão efetiva, o que é chamado de Sistema de Compilação Efetivo. Para tal, esta seção também apresenta a criação da estrutura interna de grafo associada ao MCPON efetivo, em uma versão agora efetiva do Grafo PON, e o funcionamento dela no respectivo Sistema de Compilação Efetivo. Este permite a composição de respectivo conjunto de compiladores quando é consagrado com a implementação de um ou mais geradores de códigos pertinentes.

Objetivamente, o Sistema de Compilação Efetivo é a implementação do MCPON Efetivo, por meio de algum conjunto de técnicas, as quais permitem a criação de uma ou mais linguagens, bem como um ou mais compiladores para o PON.

Nesse âmbito, a próxima seção descreve o Sistema de Compilação Efetivo naturalmente criado pelo próprio autor do MCPON. Além disso, o capítulo apresenta as evoluções das tecnologias desenvolvidas pelos pesquisadores e colaboradores do grupo de pesquisa PON, baseados nas diretrizes da versão efetiva do método MCPON e, naturalmente do Sistema de Compilação Efetivo em seus trabalhos[22]. Em geral tais trabalhos visaram essencialmente expandir a própria NOPL e o sistema de compilação associado, particularmente no tocante aos conceitos faltantes. Essa evolução colaborativa enriquece a Tecnologia LingPON 2.0 (ou Tecnologia NOPL) resultante, inclusive, por meio da construção de geradores de código associados as fases anteriores automatizadas no Sistema de Compilação Efetivo.

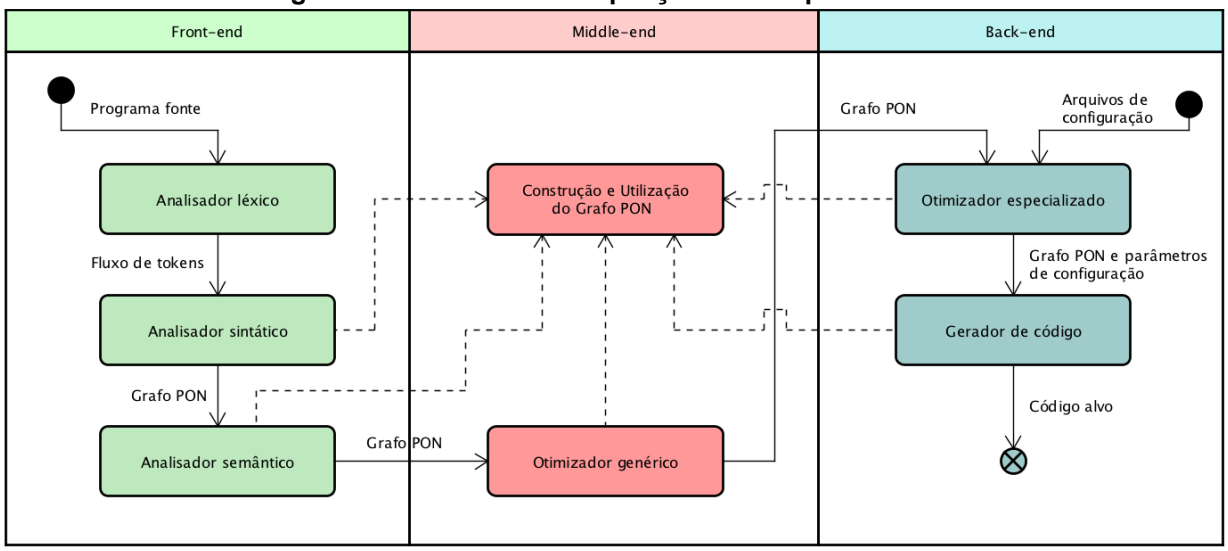
6.3.1 Sistema de Compilação Efetivo para o PON

De maneira geral, o Sistema de Compilação Efetivo para o PON foi construído seguindo as diretrizes do método MCPON em sua versão Efetiva, apresentando um *front-end* com as fases de análise (i.e., subetapas léxica e sintática da Etapa 1 do MCPON Efetivo), um *middle-end* baseado na representação intermediária Grafo PON

[22] Em tempo, é pertinente resalientar que, além do MCPON Efetivo e respectivo Sistema de Compilação Efetivo, o desenvolvimento da primeira versão da NOPL e o primeiro compilador foi também realizado pelo autor deste trabalho. A partir daí, as evoluções na NOPL e a construção de demais compiladores foram por ele acompanhados, primeiramente em coautoria e depois tutoreando os demais pesquisadores e colaboradores na aplicação de tal tecnologia.

Efetivo (Etapa 2 do MCPON Efetivo), bem como um Otimizador genérico (Subetapa 3.1 do MCPON Efetivo) e um *back-end* composto por um Otimizador especializado (Subetapa 3.2 do MCPON Efetivo), bem como pela fase de geração de código (Etapa 4 do MCPON Efetivo). Esta versão do Sistema de Compilação Efetivo é apresentada na Figura 75.

Figura 75: Sistema de Compilação Efetivo para o PON



Fonte: Autoria Própria via Diagrama de atividades em UML

Conforme apresenta a Figura 75, o Sistema de Compilação Efetivo para o PON se apresenta de forma coesa e desacoplada, no sentido de que o desenvolvedor poderia trabalhar individualmente em cada uma das fases ou, pelo menos, em camadas distintas do Sistema de Compilação Efetivo, de modo a aproveitar todo o restante que já está construído e validado. Nesse âmbito, seria possível construir uma linguagem para o PON inteiramente nova, reconstruindo apenas o *front-end* do compilador. Com isso, seria possível aproveitar todos os *N* targets já compilados para tal, sem precisar fazer nenhum ajuste nas camadas seguintes. Assim como também seria possível, e mais comum, construir um novo *target* para o PON, desenvolvendo apenas a camada de *back-end* do compilador. Com isso, seria possível aproveitar de uma linguagem própria e madura, bem como das etapas de análises e construção da representação intermediária. Nesse âmbito, de modo a apresentar em maiores detalhes as camadas do Sistema de Compilação Efetivo, as próximas subseções detalham as etapas e subetapas em questão que levaram a composição do Sistema de Compilação em pauta.

6.3.1.1 *Front-end* – Etapa 1 do MCPON Efetivo

O Sistema de Compilação Efetivo do PON foi construído de maneira a contemplar todas subetapas da Etapa 1 do Método MCPON Efetivo no âmbito do *front-end* deste sistema. Em termos de linguagem, o *front-end* desta versão foi totalmente baseado na primeira SubEtapa da Etapa 1 (Definição da linguagem). Assim sendo, contemplou-se todas as subetapas da Etapa 1 do método que são: (a) Definição das Características da Linguagem; (b) Definição das Palavras-chave e Analisador Léxico; e (c) Definição das Regras Gramaticais e Analisador Sintático.

Isto posto, o analisador léxico foi construído com base na ferramenta open-source Flex. Em tal ferramenta foram definidas todas as palavras reservadas pertinentes a linguagem NOP, bem como a identificação de IDs, números e demais elementos para uma correta interpretação dos tokens da linguagem.

A análise sintática, por sua vez, foi desenvolvida a partir da ferramenta *open-source* Bison (BISON, 2019), a qual já possui uma integração com a ferramenta Flex (FLEX, 2019), possibilitando interpretar os *tokens* extraídos pela etapa anterior e mapeá-los na análise sintática de acordo com a *BNF* da linguagem especificada na ferramenta Bison. É importante salientar que a *BNF* completa da linguagem de programação NOPL se encontra no Apêndice F.

Com base nas ferramentas Flex e Bison, os quais geram módulos com analisador léxico e analisado sintático em linguagem C, é possível então criar a integração da fase de análises (*front-end*), com a construção do Grafo PON (*middle-end*) que em termos da materialização da NOPL se constitui em uma classe em C++ (com respectiva instância ou objeto) com os dados em forma de grafo e os métodos correlatos para popular os dados do grafo em si, acessá-los, modificá-los etc.

Após a construção do Grafo PON (apresentado em maiores detalhes na próxima subseção), o processo de compilação normalmente seguiria para a definição e aplicação das regras semânticas, as quais são basicamente verificações pontuais realizadas a partir da iteração do Grafo PON.

6.3.1.2 *Middle-end* – Grafo PON e Otimizações Genéricas

O *middle-end* do Sistema de Compilação Efetivo para o PON consiste essencialmente na construção de uma *API*, ou mais precisamente, pela

implementação do padrão de projeto *Facade* com as características do Grafo PON. Esta "*API"* basicamente fornece métodos simplificados para a instanciação das entidades no Grafo PON, bem como as respectivas entidades relacionadas. Além disso, ela fornece um conjunto de métodos que facilitam a navegação e iteração no Grafo PON, de modo a facilitar a interação das demais partes constituintes do Sistema de Compilação com o próprio grafo.

Ademais, o *middle-end* consiste na integração das Etapas 1 e 2 do MCPON, onde a fase de análises (*front-end*) é totalmente integrada com a construção e navegação do Grafo PON. Justamente, essa integração é feita por meio desta "*API*", simplificando a construção do Grafo PON por meio dos métodos já devidamente mapeados para tal.

Por fim, o *middle-end* também é responsável pela criação de otimizadores genéricos independentes de *target*, subetapa da Etapa 3 do MCPON. Esta atividade também é intermediada por esta "*API*", onde é possível encontrar redundâncias ao iterar apropriadamente o grafo, permitindo o processo eliminá-las quando necessário.

6.3.1.3 *Back-end* – Otimizações Especializadas e Geração e Código

Em termos de MCPON, o *back-end* é construído essencialmente pela Etapa 4 (Geração de Código), nas suas duas subetapas: (a) iterar instâncias do Grafo PON; e (b) Construção de Geradores de Código. Em tempo, o *back-end* também se apoia na Etapa 3 (Construção de Otimizadores), mais precisamente em sua segunda subetapa: (a) Criação de Otimizadores especializados dependentes de *target*. Tal subetapa é tida como opcional, entretanto, para sua implementação o desenvolvedor poderia utilizar arquivos de configuração, seja em formato texto, xml etc. Na construção da Tecnologia NOPL em si optou-se pela interpretação de um arquivo texto em formato simples, descrevendo os parâmetros linha a linha, no padrão: nome do parâmetro seguido do respectivo valor, conforme será apresentado em 6.3.2.2.3.

6.3.2 Expansão do Sistema de Compilação Efetivo

De maneira geral, um dos principais objetivos desta tese de doutorado é validar o método MCPON por meio de seu Sistema de Compilação Efetivo. Não somente isso, mas permitir que outros desenvolvedores (especialistas e não-especialistas em PON) possam aplicar as etapas do método na prática, no âmbito de expandir o sistema como um todo.

Nesse âmbito, o método MCPON foi aplicado por um grupo de discentes (1 de graduação em bacharelados e 10 de pós-graduação stricto sensu) da instituição de afiliação (UTFPR). Isto foi estimulado, em comum acordo, no âmbito de uma disciplina stricto senso de Linguagens e Compiladores em 2018 (via PPGCA & CPGEI – UTFPR). Estes discentes, ao passo em que adquiriram conhecimentos básicos sobre a construção de linguagens e compiladores, puderam aplicar etapas do método na prática. O estudo contou com a participação de 11 pessoas, no qual cada qual pode contribuir pontualmente em partes isoladas da construção da Tecnologia NOPL, em sua maioria no âmbito de construção dos geradores de código pertinentes.

Nesse âmbito, esta seção apresenta expansões em termos de linguagem que desenvolvedores utilizaram para demonstrar a capacidade de extensibilidade do método aplicado à própria linguagem NOPL. Ademais, com o método MCPON Efetivo é possível simplificar a construção de compiladores para plataformas distintas que se apoiam das características e propriedades fundamentais do paradigma na construção de programas cada vez mais performantes, paralelizáveis e distribuíveis.

6.3.2.1 Expansão da linguagem NOPL

Apesar da linguagem NOPL se apresentar de forma mais completa que a linguagem predecessora, observou-se que o suporte à vetores permitiria maior flexibilidade na programação como um todo, assim como o conceito de *Formation Rules*, o qual havia sido implementado na versão anterior da linguagem. Nesse âmbito, permitir a facilidade de extensão da linguagem foi uma das principais preocupações com as evoluções propostas no método MCPON Efetivo. Nesse âmbito, as próximas subseções apresentam as evoluções linguísticas no âmbito da linguagem NOPL.

6.3.2.1.1 Suporte a Vetores

De maneira a suportar a implementação de vetores na linguagem NOPL, foram levantados em consideração os seguintes requisitos para tal (LAUTERT, 2018):

- A linguagem deveria suportar vetores de *Attributes* (*Integer*, *Float*, *String*, *Boolean* e *Char*).
- A linguagem deveria suportar vetores de instâncias de *FBE*.
- A linguagem deveria suportar vetores com valores de índices dinâmicos (*e.g.,* vetor[i]) e estáticos (*e.g.,* vetor[3]).
- A linguagem deveria suportar vetores com valores de índices calculados (*e.g.,* vetor[i + 1]), suportando soma e subtração.
- Regras semânticas na validação de índices dos vetores (impedir uso de índices inexistentes).

Como o Sistema de Compilação Efetivo já possuía a implementação de geração de código para várias linguagens-alvo, optou-se por não criar novas estruturas no Grafo PON, mas sim instanciar automaticamente cada entidade pertinente para popular o grafo adequadamente (LAUTERT, 2018). Nesse âmbito, o Código 64 apresenta exemplos de declarações de vetores na linguagem NOPL.

Código 64: Exemplos de declarações de vetores na linguagem NOPL

```
1  public Sensor[2] sensorVector
2
3  public Alarm[5] alarmVector
4
5  public boolean[2] xAlarm
6
7  public boolean[2] jAlarm = {true, false}
```

Fonte: Adaptado de Lautert, 2018

Conforme apresenta o Código 64, as declarações de vetores seguem o mesmo padrão de instanciação de *FBEs* (linhas 1 e 3) ou de *Attributes* (linhas 5 e 7), adicionando entre colchetes o número de elementos do vetor. É importante observar que o valor inicial das entidades é opcional, sendo definido o valor padrão em caso de falta de valor inicial. O Código 65 apresenta um exemplo de atribuição em um índice estático na linguagem NOPL.

Código 65: Exemplo de atribuição por índice estático na linguagem NOPL

```
1  private method mtSendSms
2    assignment
3      this.jAlarm[1] = true
4    end_assignment
5  end_method
```

Fonte: Adaptado de Lautert, 2018

Conforme apresenta o Código 65 a atribuição de um valor à um *Attribute* é semelhante à atribuições convencionais, com a adição do colchetes e o índice estático do elemento do vetor. Ainda, o Código 66 apresenta um exemplo de utilização dos vetores em comparações de *Premises* na linguagem NOPL.

Código 66: Exemplos de utilização dos vetores na linguagem NOPL

```
1  rule rlNormal
2    condition
3      premise prThird
4        this.jAlarm[1] == true
5      end_premise
6      or
7      premise prOne
8        sensorVector[0].atStatus == alarmVector[0].atStatus
9      end_premise
10   end_conditon
11 end_rule
```

Fonte: Adaptado de Lautert, 2018

Conforme apresenta o Código 66, as comparações em *Premises* também poderiam ser baseadas em elementos de vetores, identificados por meio de índices estáticos.

6.3.2.1.2 Suporte a *Formation Rules*

Ademais, vislumbrou-se a possibilidade de criar a funcionalidade de *Formation Rules* associada a tais vetores, a qual permitiria a repetição de *Rules* para diferentes índices do vetor. Na prática, a implementação do conceito de *Formation Rules* facilita o desenvolvimento de programas PON e, por consequência, diminui a repetição de regras semelhantes na essência do programa (LAUTERT, 2018).

A título de exemplo, considera-se um cenário de um sistema de alarme, no qual cada sensor associado teria que enviar uma notificação sob uma determinada condição. Sem o uso dos vetores, o desenvolvedor precisaria criar uma *Rule* para cada sensor. Entretanto, com base nos vetores e a funcionalidade de *Formation Rules*, seria possível escrever uma única regra de geração automática e, em tempo

de compilação, seriam instanciados as *N* entidades pertinentes para cada uma das *Rules* associadas aos sensores envolvidos. Nesse âmbito, o Código 67 apresenta um exemplo de *Formation Rule* com a definição de índices de repetição.

Código 67: Exemplos de *Formation Rules* na linguagem NOPL

```
1  formation_rule rlFireAlarm
2    index i from 0 to 1
3    index j from 0 to 1
4    condition
5      premise prOne
6        sensorVector[i].atStatus == alarmVector[j+1].atStatus
7      end_premise
8      or
9      premise prSecondFRule
10       this.jAlarm[j] == true
11     end_premise
12   end_condition
13 end_formation_rule
```

Fonte: Adaptado de Lautert, 2018

Conforme apresenta o Código 67, uma *Formation Rule* é definida por meio das palavras reservadas ***formation_rule*** e ***end_formation_rule***. Nesta estrutura é obrigatória a definição de pelo menos um índice, o qual define a quantidade de *Rules* geradas a partir desta estrutura. Ao definir dois ou mais índices, o compilador constrói automaticamente uma relação $i \times j \times \ldots$ *N* destas, permitindo gerar uma *Rule* pontual para cada combinação de todos os índices.

Tanto a implementação de vetores quanto o conceito de *Formation Rules* foram elaborados por Filipe Lautert com coautoria de Leonardo Faix Pordeus e do autor desta tese. É importante ressaltar que esta implementação teve impacto apenas nas análises léxica, sintática e semântica (Etapa 1 do método MCPON) e na subetapa 2.1 (Instanciar as entidade e popular instâncias do Grafo PON).

Ainda mais importante é observar que a expansão da linguagem em si, principalmente por meio de um incremento relevante e impactante, só implicou em mudanças no *front* e no *middle-end*, não requerendo mudanças no *back-end*. Se não houvesse um elemento desacoplante como o Grafo PON, tais mudanças demandariam esforços extras, o que implicaria na modificação de cada um dos geradores de código associados. O fato de existir o Grafo PON e de, portanto, a expansão da linguagem não ter implicado em mudanças nos geradores de código em si, é uma importante característica organizacional-desacoplante dele. Basicamente, em tais conceitos implementados (*i.e.,* Vetores e *Formation Rules*), foi possível

instanciar as entidades pertinentes adequadamente de acordo com a semântica das instruções fornecidas pela linguagem, populando finalmente, as instâncias do Grafo PON com mais entidades oriundas do próprio Grafo PON.

6.3.2.2 Geradores de Código para Plataformas Distintas

A proposta do PON em relação a composição de programas paralelizáveis e/ou distribuíveis à luz de sua teoria, motivou a implementação de diferentes materializações para as mais distintas plataformas computacionais. Conforme Seção 3.4, diversas materializações foram implementadas, geralmente em forma de *frameworks*, o que permitiu demonstrar em alguma medida as propriedades elementares do PON. Porém a propriedade elementar relacionada a facilidade de programação, particularmente orientada a descrição em alto nível dos programas para o paradigma, nem sempre foi atendida em tais materializações. Com o advento da linguagem e do sistema de compilação próprios para o PON, tornou-se possível agregar a cada uma das materializações criadas para o PON, a propriedade elementar de programação em alto nível.

Nesse âmbito, com base no estudo colaborativo apresentado anteriormente, foi atribuído um *target* específico a cada discente participante para o qual eles deveriam, cada qual, desenvolver um gerador de código específico. Na prática, os *targets* definidos para qual são os seguintes ambientes/plataformas: (a) Framework PON C++ 1.0; (b) Framework PON C++ 2.0; (c) Framework PON C++ 3.0 adaptado (Multithread & PON IP); (d) Framework PON Java; (e) Framework PON C#; (f) C++ Namespace com notificações monothread; (g) Assembly NOCA; e (h) Framework PON Erlang/Elixir. Tais contribuições são apresentadas nas seções subsequentes.

6.3.2.2.1 *Namespaces* com notificações *mono-thread*

Essa versão, em especial, foi a primeira implementação efetiva e completa do compilador para a linguagem NOPL. A implementação desta versão de geração de código se deu essencialmente por meio da Etapa 4, com posterior validação do *target* por meio da Etapa 5. Este *target* surgiu a partir da disciplina de 2018 e foi elaborada por Larissa Keiko Oshiro com coautoria de Leonardo Faix Pordeus e do autor desta tese (OSHIRO, 2018).

Em linhas gerais, Oshiro reconstruiu a estrutura orientada a *namespaces*, proposta originalmente por Athayde, conforme Seção 5.3.2.2. Tal versão foi a que apresentou os melhores resultados em termos de desempenho e legibilidade dentre os novos *targets* propostos no âmbito da LingPON 1.X. Entretanto, Oshiro precisou fazer adaptações importantes na estrutura geral da solução, uma vez que a LingPON 1.X se orientava a instâncias globais, enquanto a NOPL se baseia em um modelo holônico que apresenta níveis de composição. Em tais adaptações foi necessário se basear em uma abordagem recursiva e em profundidade.

Oshiro também naturalmente implementou os novos conceitos e características presentes na versão atual da linguagem NOPL, a qual se apresenta de forma consideravelmente mais rica em relação a seus construtos, quando comparado as versões precedentes da linguagem.

6.3.2.2.2 *Framework* PON C++ 1.0

A implementação desta versão de geração de código se deu essencialmente por meio da Etapa 4, com posterior validação do *target* por meio da Etapa 5. Este *target* surgiu a partir da disciplina de 2018 e foi elaborada por Iverson Mendes Ferreira com tutoria do autor desta tese (FERREIRA, 2018).

Para implementação em si, Ferreira reconstruiu os elementos fundamentais do paradigma de acordo com as instâncias do Grafo PON. Como a base fundamental do *Framework* se ampara nas principais entidades do modelo do PON, a transcrição foi natural e sem maiores problemas.

6.3.2.2.3 *Framework* PON C++ 2.0 e *Framework* PON C++ 3.0

A implementação destas versões de geração de código, em particular, apresentou uma contribuição importante, no sentido de implementar pela primeira vez no método MCPON a Subetapa 3.2, responsável pela criação de Otimizadores Especializados. Ademais, a implementação destes *targets* se deu por meio da Etapa 4, com posterior validação do *target* por meio da Etapa 5. Estes *targets* para a NOPL foram elaborados conjuntamente por Christian Carlos Souza Mendes e Cleverson Avelino Ferreira, com tutoria do autor desta tese (MENDES e FERREIRA, 2018).

Em suma, os *Frameworks* PON C++ 2.0 e 3.0 foram integrados a biblioteca PON IP. Para essa integração é necessário um arquivo que contenha a pré-configuração de dados para envio dos estados dos *Attributes* via rede, como endereço de IP de destino, porta para comunicação, tipo do protocolo, entre outros. Nesse caso, o compilador em si, com base na Subetapa 3.2 precisaria se adequar as configurações especializadas no *target* em questão. Essa configuração se deu por meio de um novo arquivo em formato texto, no qual são listadas as configurações apropriadas para tal (MENDES e FERREIRA, 2018).

No momento da compilação do código em NOPL, é necessário que seja realizada a importação e validação do arquivo de configuração, o qual deve estar localizado no diretório raiz do compilador. Este arquivo de configuração, conforme apresenta o Código 68, contém as informações necessárias para configurar o *Framework* em relação ao envio do estado dos atributos por meio da rede de computadores.

Código 68: Arquivo de configuração para otimizadores específicos

```
1  #Porta de conexão
2  port 33333
3
4  #Endereço de destino
5  default_dst_ip 127.0.0.1
6
7  #Protocolo de transporte (0 = UDP | 1 = TCP)
8  protocol 0
9
10 #Debug (0 = off | 1 = on)
11 debug 1
12
13 #Número de núcleos para a paralelização
14 #Configuração específica do Framework PON C++ 3.0
15 numcore 2
```

Fonte: Mendes e Ferreira, 2018

Conforme apresenta o arquivo de configuração é possível atuar de forma específica no *target* desejado por meio desses arquivos auxiliares, sem necessariamente modificar a linguagem original, a qual continua mantendo um padrão universal. Em maiores detalhes, as configurações pontuais permitem configurar adequadamente os programas gerados, como no caso do *Framework* PON C++ 3.0, permitir definir a quantidade de núcleos para a execução paralela. Outras parametrizações poderiam ser criadas, inclusive no âmbito destes *targets*, como otimizações mais pontuais, no sentido de permitir definir quais entidades deveriam executar em quais núcleos, ou no caso da distribuição, em quais nós computacionais.

6.3.2.2.4 *Framework* PON Java e C#

A implementação das versões de geração de código para os *Frameworks* PON Java e C# se deram essencialmente por meio da Etapa 4, com posterior validação do *target* por meio da Etapa 5. Tais *targets* surgiram a partir da disciplina de 2018 e foram elaboradas com tutoria do autor desta tese. A implementação de gerador de código para o *Framework* PON Java foi construída por Leonardo Trevisan Silio (SILIO, 2018), sendo este discente (então de terceiro período) de engenharia da computação da UTFPR Curitiba. A implementação de gerador de código para o *Framework* PON C#, por sua vez, foi construído por Paulo Bertoldi Renaux, com auxílio de Leonardo Trevisan Silio (RENAUX, 2018).

Para implementação em si, ambos os *frameworks* seguiram o processo de iteração do grafo, mapeando e reconstruindo fidedignamente cada uma das entidades principais na estrutura do código-alvo. É importante ressaltar que a construção de compiladores para *frameworks* normalmente é relativamente mais fácil do que para outros *targets* otimizados, justamente pelo fato de que o *framework* define o conjunto das entidades fielmente ao modelo tradicional da teoria do PON, bastando “apenas” instanciar adequadamente cada uma delas.

6.3.2.2.5 *AssemblyPON - NOCA*

A implementação desta versão de geração de código se deu essencialmente por meio da Etapa 4, com posterior validação do *target* por meio da Etapa 5. Este

target surgiu a partir da disciplina de 2018 e foi elaborada por Luiz Fernando Copetti com tutoria de Leonardo Faix Pordeus e do autor desta tese (COPETTI, 2018).

Devido a otimizações pontuais no âmbito da própria NOCA, a qual elimina a necessidade de algumas das entidades principais do PON em sua construção, Copetti questionou a praticidade do Grafo PON em relação a esta questão. Copetti relatou certa dificuldade no processo, no qual precisou "desmontar" a estrutura do grafo para remapear as entidades de acordo com o *target* visado, para só então reconstruir o código alvo adequadamente (COPETTI, 2018).

É importante salientar que algumas destas dificuldades também foram apontas inicialmente pelos desenvolvedores da LingPON 1.X e algumas delas foram sanadas no modelo atual. Entretanto, é sempre importante coletar novos *feedbacks*, essencialmente em termos de novas materializações que tratam o PON de maneira, por vezes, diferenciada, e assim, decidir por fornecer novos métodos no Grafo PON que auxilie a busca pelos elementos de maneira mais prática, conforme o caso.

Nesse sentido, uma propriedade interessante do Grafo PON é sua natural extensibilidade dado sua natureza de implementação orientada a objetos, à luz do princípio de coesão e desacoplamento ou, outramente dito, modularidade.

6.3.2.2.6 PON-HD 1.0

A implementação desta versão de geração de código se deu essencialmente por meio da Etapa 4, com posterior validação do *target* por meio da Etapa 5. Este *target* surgiu a partir da disciplina de 2018 e foi elaborada por Edemar A. Lanes Junior com tutoria do autor desta tese (LANES JR, 2018).

Lanes Jr relatou que "em relação à utilização da LingPON, pode-se ressaltar que, após o correto entendimento da estrutura, foi possível operar com ela de maneira bastante prática". Lanes Jr porém relatou algumas dificuldades em relação a localização de algumas propriedades de determinadas entidades no Grafo PON, bem como a falta de alguns conceitos que não estavam implementados no estado da técnica naquela época (LANES JR, 2018); questões estas que foram resolvidas em um segundo momento.

6.3.2.2.7 *Framework* PON Erlang/Elixir

A implementação desta versão de geração de código se deu essencialmente por meio da Etapa 4, com posterior validação do *target* por meio da Etapa 5. Este *target* surgiu a partir dos esforços de Fabio Negrini com tutoria do autor desta tese (NEGRINI, 2019). A escolha deste *target* se deu principalmente porque a máquina virtual Erlang permite programação em alto nível, compilação e execução concorrente. Essa concorrência se dá de forma natural em função das características de desacoplamento da linguagem Erlang que, aliada à arquitetura multicore da máquina virtual BEAM disponibilizada pela arquitetura Erlang/OTP, permitem o uso de seus métodos de balanceamento (CESARINI e THOMPSON, 2009; NEGRINI, 2019).

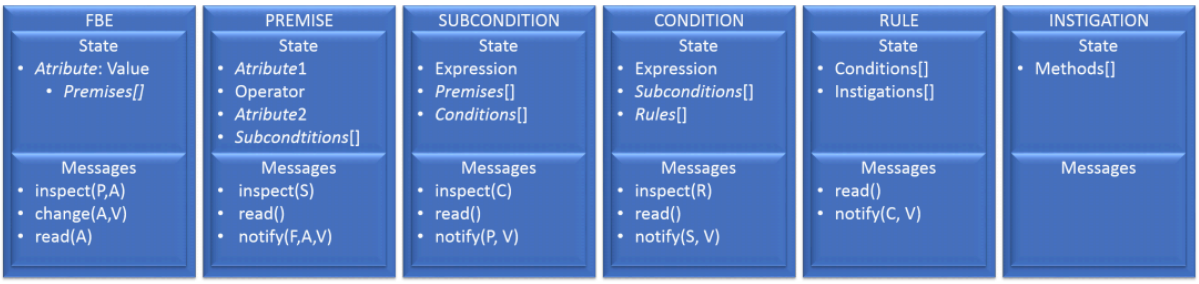
A linguagem Erlang possui seu próprio *heap* e pilha e cada processo é executado em seu próprio espaço de memória. Processos não podem interferir uns nos outros inadvertidamente, como é comum em outros modelos de *multithreading*. A abordagem tradicional é condicionada a *deadlocks* e outros problemas comuns à programação concorrente. No Erlang, porém, os processos se comunicam entre si via passagem de mensagens, onde a mensagem pode ser qualquer valor de dados. A passagem de mensagens é assíncrona, portanto, uma vez que uma mensagem é enviada, o processo pode continuar o processamento.

De fato, as mensagens são recuperadas da caixa de correio (*i.e.,* fila de mensagens recebidas pelo processo) do processo seletivamente, portanto, não é necessário processar mensagens na ordem em que são recebidas. Isso torna a simultaneidade mais robusta, principalmente quando os processos são distribuídos entre diferentes computadores e a ordem em que as mensagens são recebidas dependem das condições da rede ambiente (CESARINI e THOMPSON, 2009; NEGRINI, 2019).

Em termos de paradigma, a linguagem Erlang segue os princípios do Paradigma Funcional (PF) associado com o Paradigma Orientado a Atores (POA), sendo que o POA pode ser visto como um sistema de agentes. Assim, em alguma medida, o caminho para o aproveitamento da arquitetura concorrente Erlang passa pela aderência dos elementos do paradigma PON, mais precisamente no tocante a processos com estados que se comunicam por meio de mensagens. Nesse âmbito, Negrini fez uma modelagem de cada uma das entidades PON principais na forma de micro atores. A Figura 76 apresenta um modelo alto nível da tradução dos elementos

PON em atores, de maneira que estes possam ser codificados e transformados em módulos/atores Erlang (NEGRINI, 2019).

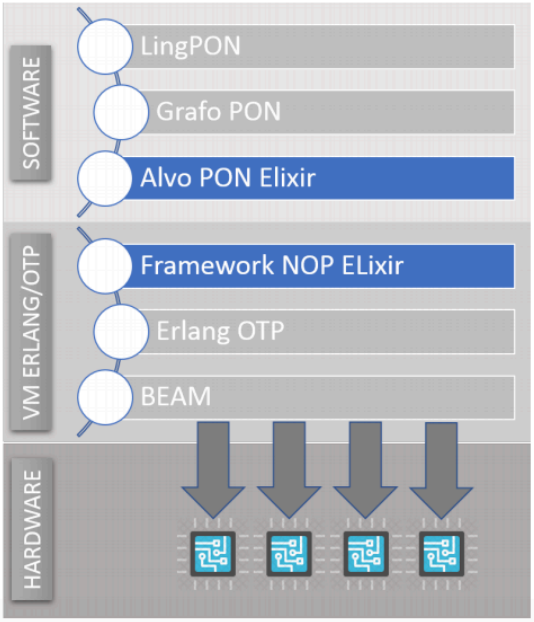
Figura 76: Modelagem os elementos PON em POA



Fonte: Negrini, 2019

Segundo Negrini (2019), após o mapeamento dos elementos do PON para elementos com comportamentos de atores, seguiu-se a etapa de codificação dos mesmos. Nesta etapa optou-se pela utilização da linguagem Elixir. Esta foi escolhida por estar totalmente imersa na plataforma Erlang e ainda disponibilizar mecanismos essenciais para a programação estruturada (NEGRINI, 2019). Na prática, Negrini construiu um *Framework* PON Erlang/Elixir e posteriormente adaptou a geração de código do sistema de compilação para tal. Esta integração é apresentada na Figura 77, a qual ilustra basicamente a arquitetura da solução proposta.

Figura 77: Arquitetura da compilação para Erlang/Elixir



Fonte: Negrini, 2019

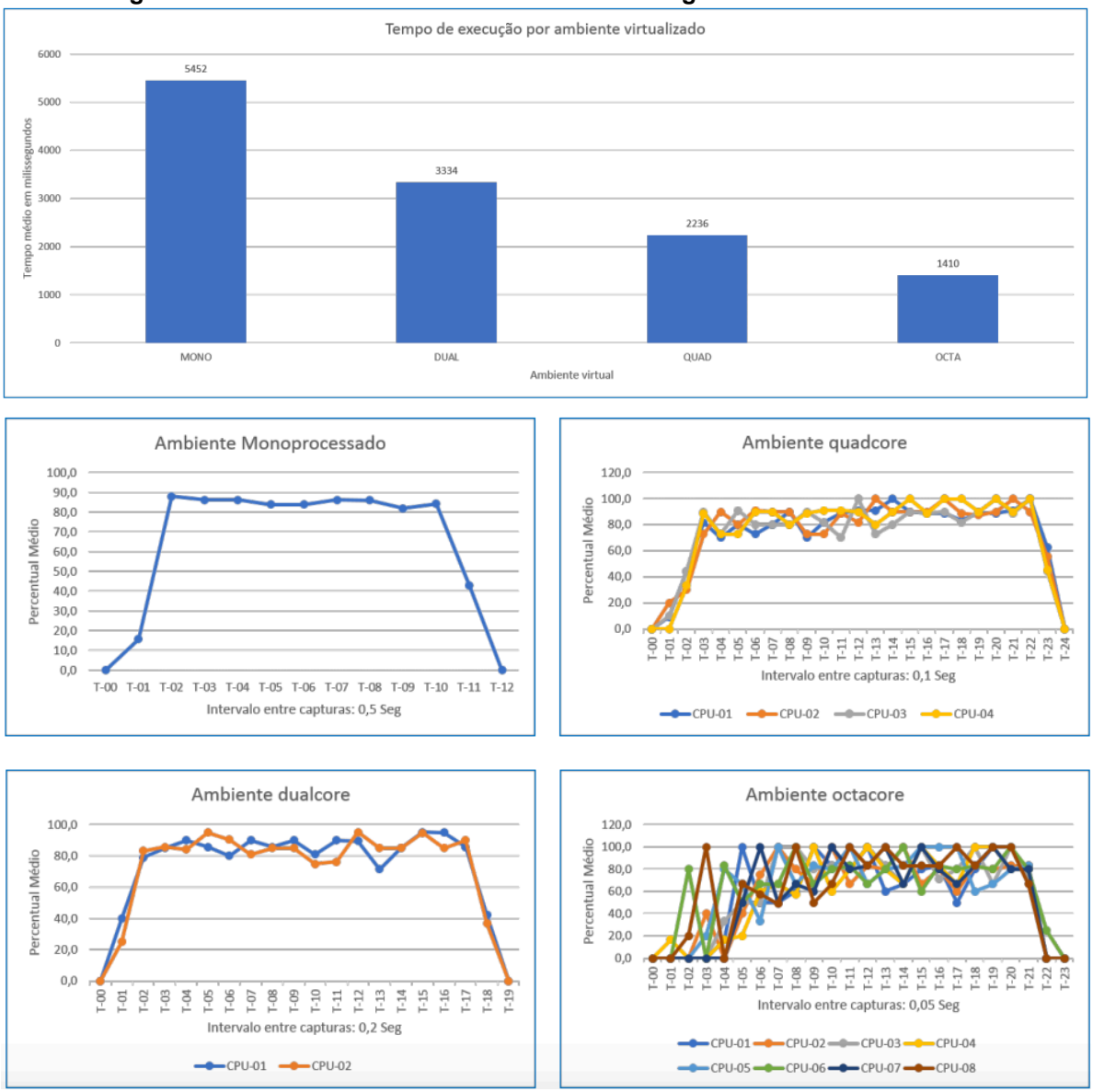
É importante observar que Negrini inicialmente construiu o *Framework* PON Erlang/Elixir à luz da teoria do PON e posteriormente fez a integração/geração de código para este a partir do Sistema de Compilação do MCPON Efetivo. Para esta integração, Negrini relatou ter gasto cerca de 10 horas de trabalho no total (NEGRINI, 2019). Esta atividade compreendeu a identificação das entidades do grafo para sua conversão em elementos do *framework* e suas ligações devidamente alinhadas. O resultado final é uma arquitetura PON traduzida para atores sob a plataforma Erlang permitindo assim o total aproveitamento da capacidade de concorrência da máquina virtual BEAM.

Para avaliar a capacidade de processamento concorrente *multicore* no *target* criado, Negrini propôs como experimento a implementação de um programa para Controle de Trânsito Automatizado (CTA). O objetivo deste programa é simular um controle de trânsito em uma matriz i x j de semáforos. O cenário consiste basicamente em um conjunto de semáforos para as ruas horizontais e um conjunto de sinais dos semáforos para as ruas verticais. Este ambiente foi reproduzido para um conjunto de dez quadras horizontais por dez quadras verticais totalizando 100 semáforos. Depois foram simulados ciclos de 2000 segundos em sequência para todos os semáforos. A partir desses experimentos com tal ferramental foi possível avaliar a eficiência da solução. Basicamente, foi possível demonstrar a transparência de paralelizar a execução do programa por meio desta, conforme apresenta os resultados ilustrados na Figura 78 (NEGRINI, 2019).

Nos resultados é possível verificar que, conforme aumenta-se o número de núcleos para um mesmo programa PON, o tempo de execução é reduzido ao passo que permanecem todos os núcleos balanceados durante todo o processamento do programa. Com base em tais experimentos foi possível demonstrar que o *framework* permite a paralelização transparente e eficiente dentro da arquitetura proposta. Em suma, este aproveitamento alcançado com programação alto nível e estruturada, sem interferência ou conhecimento do programador em ambientes *multicore*, se torna possível graças ao desacoplamento da linguagem LingPON aliado à arquitetura concorrente Erlang. Somente com Erlang, tal concorrência somente seria possível

com o desenvolvimento de alguma técnica de programação paralela, algo que ficou totalmente transparente graças à camada LingPON[23].

Figura 78: *Benchmarks* do *Framework* PON Erlang em ambientes *multicore*



Fonte: Negrini, 2019

6.3.3 Validação do processo

Após a implementação de um *target* específico, a última etapa do método MCPON (Etapa 5), consiste fundamentalmente na validação do processo de

[23] Existe outro trabalho em andamento, mais precisamente um *Framework* PON orientado a atores/agentes na linguagem/tecnologia AKKA. Martini ainda não integrou esta materialização no âmbito da Tecnologia NOPL (MARTINI, 2018).

compilação como um todo. Tal etapa, basicamente, contempla as validações dos *targets* gerados por meio de um conjunto de pequenos programas PON, em forma de grafos previamente criados (*i.e.*, programas interpretados e mapeados em função do Grafo PON). Tais instâncias do Grafo PON possuem o intuito de validar as principais características do PON por meio de testes minimalistas e pontuais que visam medir a integridade e abrangência de conceitos implementados no compilador (*i.e.*, gerador de código) criado. Além disso, esta etapa também deveria considerar programas sintaticamente e semanticamente errados, de modo a validar a efetividade das regras de validação do compilador, a ponto de falhar e interromper o processo de compilação apropriadamente.

Nesse sentido, o autor deste trabalho criou alguns pequenos programas para validar individualmente e coletivamente cada um dos conceitos do PON. A título de exemplo, o Código 69 apresenta um dos programas criados para os testes de integridade.

Código 69: Programa para teste de integridade em linguagem NOPL

```
 1  fbe Main
 2
 3    private Boolean atStatus = false
 4
 5    private method mtChange
 6      assignment
 7        this.atStatus = true
 8      end_assignment
 9    end_method
10
11    rule rlChange
12      condition
13        premise
14           this.atStatus == true
15        end_premise
16      end_condition
17      action
18        instigation
19          call this.mtChange
20        end_instigation
21      end_action
22    end_rule
23
24    main
25      this.atStatus = true
26    end_main
27
28  end_fbe
```

Fonte: Autoria Própria

Basicamente, esse exemplo em si testa um programa mínimo em PON, composto por um único *FBE* (*i.e., FBE Main*), um *Attribute* (*i.e., atStatus*), um *Method* (*i.e., mtChange*) e uma *Rule* (*i.e., rlChange*). Nesse programa também é utilizado o bloco *main*, para a inicialização do programa. Com base nesse programa mínimo é possível validar o *target* quanto a sua capacidade de gerar um programa igualmente mínimo na linguagem-alvo.

Além deste programa citado, existe um conjunto de outros pequenos programas que validam basicamente os principais conceitos fundamentais do PON. De maneira a apresentar os conceitos fundamentais do PON em relação aos *targets* da Tecnologia NOPL, a Tabela 10 apresenta a integralidade dos *targets* com base nos conceitos fundamentais do PON.

Tabela 10: Conceitos fundamentais contemplados nos *targets* da NOPL

Materialização / Conceitos	NOPL para Software							NOPL para Hardware	
	NAMESP MONO 2018	FW C++ 1.0 2018	FW C++ 2.0 2013	FW C++ 3.0 2018	FW JAVA 2018	FW C# 2018	FW ERLANG 2019	NOPL NOCA 2018	NOPL PONHD 2018
Reatividade das entidades	✓	✓	✓	✓	✓	✓	✓	✓	✓
Escalonamento de *Rules*		✓	✓	✓	✓	✓		✓	
Estratégias de resolução de conflito		✓	✓	✓	✓	✓		✓	
Compartilhamento de entidades	✓	✓	✓	✓	✓	✓	✓	✓	✓
Regras de formação	✓	✓	✓	✓	✓	✓	✓	✓	✓
Propriedades reativas dos *Attributes*			✓	✓					
Master Rule			✓	✓					
Entidades impertinentes			✓	✓					
FBE Rules	✓	✓	✓	✓	✓	✓	✓	✓	✓
FBE agregador	✓	✓	✓	✓	✓	✓	✓	✓	✓
Vetores	✓	✓	✓	✓	✓	✓	✓	✓	✓

Fonte: Autoria Própria

Em termos de conceitos fundamentais contemplados nos *targets* da NOPL, houve um certo avanço em relação às versões precedentes da linguagem. Muitos dos conceitos na NOPL são implícitos graças às evoluções no MCPON e no Grafo PON

Efetivos, como o caso de *FBE Rules* e *FBE* agregador que são basicamente a organização holônica definida para esta nova linguagem.

Em relação aos conceitos de Regras de Formação e Vetores, os mesmos foram implementados de forma implícita, uma vez que foram tratados a nível de construção do Grafo PON. O compartilhamento de entidades também é considerado como um conceito implícito, uma vez que otimizadores genéricos podem ser construídos com o objetivo de eliminar qualquer redundância na estrutura do grafo, associando as conexões apropriadamente.

Isto posto, os *Frameworks* C++ PON 2.0 e 3.0 foram as primeiras materializações a apresentar todos os conceitos do PON na íntegra. É importante ressaltar que todos os demais *targets* também poderiam implementar os conceitos, bastando apenas contemplar explicitamente os conceitos faltantes em suas construções.

Ainda, como complemento aos testes de validação, sugere-se a compilação de programas completos, os quais também podem estar mapeados na forma de instâncias de grafos completos. Assim como Negrini aplicou os experimentos no programa CTA (Controle de Trânsito Automatizado), outros discentes estão aplicando e desenvolvendo este programa e uma versão mais robusta do programa Sensores, de modo a testar os *targets* disponíveis na Tecnologia NOPL. Alguns destes trabalhos já se encontram mapeados na página de relatórios técnicos do PON (PON, 2019).

6.4 CONSIDERAÇÕES SOBRE A TECNOLOGIA NOPL

A Tecnologia NOPL (ou Tecnologia LingPON 2.0), apresentada ao longo deste capítulo, contempla a proposta inicial da linguagem NOPL e suas evoluções linguísticas posteriores. Além disso, a Tecnologia NOPL em si, incorpora o próprio MCPON e Grafo PON implementados de maneira a construir o Sistema de Compilação Efetivo para o PON. Nesse âmbito, alguns *targets* foram implementados no âmbito de validar a tecnologia como um todo, conforme apresentado na Seção 6.3.2.2. Tais *targets* cumpriram seu objetivo de demonstrar a viabilidade de transformar programas fonte em linguagem NOPL para seus respectivos código-alvo de maneira a contemplar efetivamente a primeira propriedade elementar do PON (programação em alto nível), conforme apresenta Tabela 11.

Tabela 11: Propriedades elementares contempladas nos *targets* da NOPL

Materialização / Propriedade	NOPL para Software							NOPL para Hardware	
	NAMESPMONO 2018	FW C++ 1.0 2018	FW C++ 2.0 2013	FW C++ 3.0 2018	FW JAVA 2018	FW C# 2018	FW ERLANG 2019	NOPL NOCA 2018	NOPL PONHD 2018
Prog. Alto nível	✓	✓	✓	✓	✓	✓	✓	✓	✓
Paralelismo				✓			✓	✓	✓
Distribuição			✓	✓					
Desempenho	✓						✓	✓	✓

Fonte: Autoria Própria

A tabela mostra uma importante contribuição em termos de materializações para software. O *target* para *Framework* PON Erlang/Elixir apresentou de forma transparente a paralelização das entidades do PON, as quais foram aplicadas sob uma plataforma orientada a atores e troca de mensagens. É importante ressaltar que tal materialização também poderia considerar a distribuição do processamento, bastando para tal implementar e testar em um ambiente distribuído dado que isso é transparente em Erlang/Elixir.

Em relação ao *target Framework* PON C++ 3.0 foi possível associar a facilidade de programação em alto nível com as já presentes características de paralelismo por meio de suas implementações baseadas em *threads*, assim como também pela distribuição via PON IP. A principal contribuição deste *target*, em especial, foi a possibilidade de configurar o programa de maneira específica sem a necessidade de alterações na linguagem principal. Este *target,* porém, não se apresenta de forma otimizada em termos de desempenho, justamente por se basear em uma implementação orientada por objetos dinâmicos e listas de ponteiros, conforme Capítulo 3.

Em termos de aplicação do método MCPON no âmbito de construção dos *targets* para a linguagem NOPL, a Tabela 12 apresenta a evolução da utilização do método nas principais materializações implementadas.

Tabela 12: Subetapas do método MCPON contempladas nos *targets* da NOPL

Materialização / Subetapas	NOPL para Software							NOPL para Hardware	
	NAMESPMONO 2018	FW C++ 1.0 2018	FW C++ 2.0 2013	FW C++ 3.0 2018	FW JAVA 2018	FW C# 2018	FW ERLANG 2019	NOPL NOCA 2018	NOPL PONHD 2018
Subetapa 1.1 Definição das características da linguagem	✓	✓	✓	✓	✓	✓	✓	✓	✓
Subetapa 1.2 Definição das palavras-chave e analisador léxico	✓	✓	✓	✓	✓	✓	✓	✓	✓
Subetapa 1.3 Definição das regras gramaticais e analisador sintático	✓	✓	✓	✓	✓	✓	✓	✓	✓
Subetapa 2.1 Instanciar as entidade e popular instâncias do Grafo PON	✓	✓	✓	✓	✓	✓	✓	✓	✓
Subetapa 2.2 Construir a integração das análises com o Grafo PON	✓	✓	✓	✓	✓	✓	✓	✓	✓
Subetapa 2.3 Definição das regras semânticas e analisador semântico	✓	✓	✓	✓	✓	✓	✓	✓	✓
Subetapa 3.1 Criação de otimizadores genéricos independentes de target	✓	✓	✓	✓	✓	✓	✓	✓	✓
Subetapa 3.2 Criação de otimizadores especializados dependentes de target			✓	✓					
Subetapa 4.1 Iterar instâncias do Grafo PON	✓	✓	✓	✓	✓	✓	✓	✓	✓
Subetapa 4.2 Construção de geradores de código	✓	✓	✓	✓	✓	✓	✓	✓	✓
Subetapa 5.1 Testes de integridade	✓	✓	✓	✓	✓	✓	✓	✓	✓
Subetapa 5.2 Compilação de programas completos	✓	✓	✓	✓	✓	✓	✓	✓	✓

Fonte: Autoria Própria

Conforme apresenta a Tabela 12, o MCPON foi aplicado de forma completa na construção da NOPL e sistema de compilação associado. Os *targets* em si, aproveitam boa parte das subetapas comuns entre as implementações (marcadas como ‘comum’ na tabela). Na prática, a fase de análises e construção dos grafos

especializados são realizadas por meio do sistema de compilação geral de forma genérica. As etapas específicas, geralmente voltadas a geração de código e validação dos *targets* gerados, ficaram a cargo de cada um dos desenvolvedores, conforme Seção 6.3.2.2. É importante observar que os *Frameworks* PON C++ 2.0 e 3.0 foram os primeiros a contemplar efetivamente todas as etapas do método, uma vez que implementaram a subetapa 3.2, responsável pela criação de otimizadores especializados no *target*.

Muito embora, tenha sido possível demonstrar a viabilidade do método MCPON na criação da linguagem NOPL, espera-se que o método MCPON permita que novas linguagens para o PON sejam criadas. Tais linguagens poderiam seguir um aspecto mais purista e didático, de modo a instruir novos adeptos ao paradigma, como se apresenta a NOPL ou, por outro lado, seguir um padrão mais direto e pontual, apresentando justamente maior produtividade, especialmente voltadas para especialistas em PON.

Em termos de compilação, o método prevê que novos *targets* sejam criados e que cada qual se apresente da maneira mais apropriada possível (tanto quanto a plataforma visada permitir), com o objetivo primário de suprir a crescente demanda por software otimizado, paralelizável e distribuível, conforme apresenta o Capítulo 1.

Os *targets* aqui apresentados já conseguiram demonstrar, em alguma medida, as propriedades elementares do PON e a aplicação de seus conceitos fundamentais em suas essências. A efetividade de uma materialização para o PON está intimamente ligada à sua construção. O desenvolvedor interessado em construir uma materialização efetiva deveria buscar por técnicas e linguagens que permitem atingir bons desempenhos, e associado a organização coesa e desacoplada das entidades, buscar implementar eficientemente a paralelização na plataforma visada, bem como utilizar de protocolos e técnicas de comunicação em rede para alcançar também a distribuição efetiva.

CAPÍTULO 7

CONCLUSÃO E TRABALHOS FUTUROS

Este capítulo apresenta as conclusões da tese de doutorado, à luz dos objetivos propostos no Capítulo 1 (Introdução), bem como em todo o trabalho desenvolvido nos esforços doutorais ao longo desses últimos anos, os quais foram apresentados neste presente documento de tese. Ainda, o presente capítulo naturalmente discorre sobre um conjunto de vislumbres vis-à-vis às contribuições aportadas pela corrente tese, os quais servem de um elenco de trabalhos futuros.

7.1 CONCLUSÃO

Esta seção apresenta as conclusões obtidas deste trabalho. Resume-se, inicialmente, às realizações à luz da metodologia utilizada na tese. Em seguida, são apresentadas as principais contribuições desta tese. Por fim, são apresentadas as conclusões desta tese.

7.1.1 Realizações à luz da Metodologia Utilizada

A metodologia utilizada no desenvolvimento deste trabalho foi definida em etapas e refinamentos destas, conforme segue. A primeira etapa consistiu basicamente na escolha do tema da tese. Inicialmente, vislumbrou-se na possibilidade de construir uma materialização efetiva para o PON, com base em uma linguagem e um compilador específicos para ele. Este tema foi apresentado pelo autor deste trabalho ao Programa de Pós-graduação em Engenharia Elétrica e Informática Industrial (CPGEI) como proposta de doutoramento.

Em seguida, efetuou-se uma revisão bibliográfica focada principalmente na pesquisa já existente sobre o PON e em técnicas para a construção de linguagens e compiladores. Com base nestes estudos de viabilidade, esta revisão foi complementada inicialmente com esforços de pesquisa, durante os quais foi implementado uma versão prototipal de uma linguagem e compilador para o PON. O processo de construção desta implementação contribuiu para o vislumbre de uma versão preliminar do método proposto nesta tese, chamado mais recentemente de

MCPON Preliminar, bem como sua materialização chamada atualmente de Sistema de Compilação Preliminar.

Naturalmente, depois do MCPON preliminar, à luz de experimentações, observações e refinos, alcançou-se o MCPON Efetivo, o qual foi apresentado efetivamente como método no Capítulo 4. Em tempo, esse reenquadramento da solução proposta como método e respectiva materialização na forma de Sistema de Compilação, inicialmente chamada apenas de “linguagem e compilador do PON”, ocorreu após apresentação da qualificação de doutoramento, cujo manuscrito é referenciado como Ronszcka (2018), no qual grande parte da experimentação com o agora chamados MCPON e Sistema de Compilação Preliminares já era relatada.

Nesse sentido, de fato, realizou-se estudos de experimentação e observação, nos quais o MCPON Preliminar e seu respectivo Sistema de Compilação Preliminar foram aplicados em alguns trabalhos do grupo de pesquisa do PON, tudo com a devida tutoria e, em alguns casos, coautoria do proponente da solução. Este conjunto de esforços resultou em uma coleção de linguagens e compiladores chamados atualmente de Tecnologia LingPON 1.X que permitiram compor experimentos, conforme detalhado no Capítulo 5. Por meio destes experimentos e observações decorrentes, foi possível coletar dados sobre a aplicação da solução, possibilitando uma compreensão refinada sobre a mesma, principalmente com base em eventuais dificuldades que os envolvidos apresentaram.

Após esta etapa, organizou-se um grupo focal exploratório e confirmatório com especialistas ou conhecedores em PON para avaliar a qualidade e efetividade do proposto Sistema de Compilação Preliminar, bem como do MCPON Preliminar em si. Em tal grupo, ao mesmo tempo, foram propostas melhorias ao método MCPON apresentado, bem como foram instigados alguns questionamentos relevantes sobre a aplicação do método na criação e evolução das linguagens e compiladores particulares ao PON. Este grupo focal contou com a participação de 10 participantes, além do autor deste trabalho. Como resultado, houve importantes vislumbres no estado da arte do método e sugestões técnicas para o sistema de compilação em si.

Com base em resultados técnicos e práticos, aliados à questionamentos do grupo de pesquisa do PON como um todo, refinou-se o objeto de pesquisa desta tese. Como resultado, o foco principal da pesquisa se voltou para o desenvolvimento do método MCPON Efetivo, bem como um respectivo Sistema de Compilação Efetivo, com o objetivo de nortear o desenvolvimento de materializações efetivas para o

paradigma em questão. Tais materializações efetivas são decorrentes linguagens e compiladores para o PON em plataformas distintas, buscando coerência entre eles e a preservação das propriedades elementares e características do PON tanto quanto cada plataforma permitir.

Por fim, o método MCPON e o Sistema de Compilação Efetivos propostos nesta tese foram conjuntamente aplicados por um grupo de discentes (de graduação em bacharelados e pós-graduação stricto sensu) da instituição (UTFPR). Estes discentes, ao passo em que adquiriram conhecimentos básicos sobre a construção de linguagens e compiladores, puderam aplicar etapas do método na prática. O estudo contou com a participação de 11 pessoas, no qual cada um pode contribuir pontualmente em partes isoladas da construção da atualmente chamada Tecnologia LingPON 2.0, usualmente chamada apenas de Tecnologia NOPL.

A Tecnologia NOPL envolve, naturalmente, o Sistema de Compilação Efetivo, um conjunto de linguagens decorrentes que, por hora, conta com uma linguagem em si chamada de NOPL, bem como um conjunto associado de geradores de código para alvos/plataformas distintas que por hora são os apresentados no Capítulo 6. Naturalmente, cada gerador de código se constitui em um compilador efetivo para a plataforma alvo, quando associado às partes comuns e compartilhadas do Sistema de Compilação Efetivo como um todo. Nesse âmbito, este conjunto de compiladores permite que a(s) linguagem(ns) para o PON sejam *multi-target* no sentido de traduzirem código-fonte para código-alvo para plataformas distintas.

7.1.2 Principais contribuições da tese

A principal contribuição desta tese é a definição do método MCPON, o qual é fortemente baseado no Grafo PON, com igual relevância em termos de contribuições desta tese. O MCPON foi concebido especificamente para nortear a construção padronizada de materializações para o PON, à luz de técnicas de compilação próprias a este paradigma. Basicamente, o método MCPON norteia o processo completo de construção de tais materializações, desde a definição de uma ou mais linguagens de programação próprias ao PON, incluindo o processo padronizado para tradução de código-fonte destas linguagens para uma representação intermediária, nomeadamente Grafo PON, até a construção do código-alvo final para uma ou mais plataformas, as quais podem ser distintas. Com base nesta contribuição, considera-

se que os principais problemas que dificultavam a construção de materializações efetivas e consistentes para o PON em plataformas distintas a partir de uma mesma linguagem (ou conjunto de linguagens) foram equacionados, cumprindo assim o objetivo principal da tese.

Em tempo, o MCPON possibilita que todo o processo seja norteado por meio de um elemento balizador, o Grafo PON. De maneira geral, o Grafo PON é uma inovação no processo de compilação para o PON pois, além de permitir representar cada programa PON com suas entidades e relações por notificação, possibilita que sejam construídos programas-alvo para diferentes plataformas (*i.e.,* linguagens, paradigmas e/ou arquiteturas computacionais), sem a necessidade de adaptar a estrutura do grafo gerado para plataformas específicas. Portanto, o MCPON possibilita a construção de compiladores que apresentem integração e compatibilidade uns com os outros, possibilitando a criação de linguagens distintas para o PON que se baseiam na mesma representação intermediária, no caso o Grafo PON, assim como é possível criar geradores de código para plataformas distintas com base nesta mesma representação intermediária.

Em suma, as características desacoplantes do Grafo PON, no final das contas, possibilitam a interligação entre *N* linguagens e *M* compiladores distintos, proporcionando uma integração entre estes, de modo que novas linguagens ou novos compiladores, aproveitariam todo um sistema completo de validações e otimizações comuns neste ambiente. Neste sentido, por exemplo, um novo gerador de código para um *target* específico pode ser testado via programas prontos ou programas de teste desenvolvidos em diferentes linguagens próprias para o PON, uma vez que esses programas estariam representados como instâncias do próprio Grafo PON, com todas as análises, validações e otimizações pertinentes já contempladas a tal forma de representação.

É pertinente recordar que o MCPON com seu Grafo PON foram implementados na forma de Sistema de Compilação em duas versões (cf. seção relatada na seção anterior), a partir do qual foi possível criar linguagens para o PON, geradores de código para plataformas distintas, alcançando compiladores que demonstram a factibilidade do método, bem como a factibilidade dessas propriedades dele aqui dissertadas. Na verdade, cada Sistema de Compilação permitiu avanços no estado da técnica e da arte do PON, o que se constituem em contribuições paralelas e complementares decorrentes do próprio MCPON.

Neste âmbito, um conjunto de linguagens de programação foram propostas, baseadas no método proposto, chamadas de LingPON 1.X, culminando subsequentemente em uma mais completa denominada LingPON 2.0 ou simplesmente NOPL. A NOPL apresenta melhorias em relação a LingPON 1.X, principalmente no tocante a orientar a construção de programas PON por meio da relação todo-parte, à luz do princípio de sistemas holônicos. Além disso, a NOPL facilita, em termos linguísticos, a integração do código PON com códigos outros, de programas legados, inclusive de outras linguagens ou paradigmas. A linguagem NOPL se apresenta de forma consideravelmente rica em construtos e é, sobretudo, expansível à luz do Sistema de Compilação e, portanto, do MCPON. Isto ficou explícito quando a linguagem foi expandida para contemplar vetores (Seção 6.3.2.1.1) e o conceito de *Formation Rules* (Seções 3.6.3 e 6.3.2.1.2), cuja solução no seio de Sistema de Compilação posto foi apenas a instanciação apropriada de mais entidades do PON seguida da população delas em instâncias do GrafoPON.

Outro avanço importante deste trabalho via Sistemas de Compilação foi a criação de um ambiente de compilação híbrido, o qual se ampara no método MCPON, permitindo reunir uma coleção de geradores de código, voltados a plataformas distintas. Cada gerador de código integrado com o Sistema de Compilação forma um novo compilador. Esses compiladores elaborados à luz do MCPON permitiram, conjuntamente, melhor demonstrar que as propriedades elementares do PON são fidedignas, usando exemplos que são reaproveitáveis entre os compiladores, novamente graças ao conceito de Grafo PON proposto.

Neste contexto dado, sejam os programas-alvo definidos para executar de forma monoprocessada, paralela ou distribuída, cada qual pode ser gerado a partir da mesma instância do Grafo PON que representa um dado programa implementado em NOPL, obviamente desde que os respectivos geradores de código estejam apropriadamente compostos. Esse fato por si só, representa um pertinente avanço no âmbito de desenvolvimento de software com PON, uma vez que criar programas que executam de forma paralela ou distribuída, é tida como uma tarefa relativamente complexa nos paradigmas usuais (conforme tratado na Seção 2.1). Isto posto, a tese permitiu demonstrar mais categoricamente as possibilidades que o PON enseja a partir dos geradores de códigos e respectivos experimentos criados na sua alçada.

7.1.3 Principais conclusões da tese

O fechamento deste trabalho permitiu responder adequadamente e pontualmente aos questionamentos levantados no Capítulo 1. Tais questionamentos são aqui relembrados para fins de fluidez das explicações conclusivas:

1) É possível criar um conjunto de conceitos e técnicas de compilação apropriadas para o PON?

A resposta para tal questão é positiva. Primeiramente, o modelo conceitual do paradigma em torno de suas entidades reativas e colaborativas possibilitou explorar mais adequadamente o paralelismo e a distribuição em função de seu desacoplamento natural enquanto modelo computacional. Isto posto, tal desacoplamento permitiu vislumbrar uma nova forma de pensar nos artefatos de sistemas de compilação como um todo e, por consequência, também apoiou a concepção do Grafo PON à luz da teoria geral do paradigma. Conjuntamente ao advento do Grafo PON, outras técnicas particulares permitiram compor cada uma das etapas do método MCPON, reaproveitando técnicas usuais em etapas que assim ensejavam. Este arranjo todo se dá de maneira tal que permite definir uma nova forma, quiçá teoria, de estruturar a construção de linguagens e compiladores apropriados para o PON.

2) É possível criar materializações para o PON de forma padronizada e consistente?

A resposta para tal questão é positiva. Com base no método MCPON e, particularmente, em sua implementação na forma de Sistema de Compilação para o PON, foi possível demonstrar que é factível criar diferentes *targets*, portanto, materializações, para plataformas distintas que atendam certos critérios particulares ao PON. Além disso, tais *targets* apresentam consistência enquanto compartilham de forma implícita boa parte dos conceitos implementados no âmbito do próprio Grafo PON. Esta uniformidade ajuda a eliminar a possibilidade de erros nas materializações, pelo menos, nas partes comuns e compartilhadas, uma vez que estas estejam devidamente testadas e maduras. Além disso, a última etapa do MCPON visa justamente validar os

targets **desenvolvidos, por meio de pequenos programas e mesmo programas completos. Por fim, a padronização é inerente ao próprio MCPON, uma vez que todo o processo de criação das materializações é orientado pelo próprio Grafo PON.**

3) É possível integrar as colaborações de diversos desenvolvedores de linguagens e compiladores do PON em um único sistema de compilação voltado para plataformas distintas?

A resposta para tal questão é positiva. Tanto na versão preliminar do Sistema de Compilação (cf. Capítulo 5) quanto na versão efetiva do mesmo (cf. Capítulo 6), diversos desenvolvedores participaram da expansão das linguagens particulares ao PON, bem como na criação de compiladores (geradores de código) distintos para o paradigma, tanto no âmbito de linguagens quanto no de plataformas distintas. O fato deste Sistema de Compilação se apoiar fortemente no MCPON, possibilita que todas as contribuições individuais sejam somadas no cerne deste sistema único e uniforme de compilação para o PON, permitindo assim, a evolução incremental do sistema como um todo. Particularmente, na versão efetiva do Sistema de Compilação, essa afirmativa positiva se torna mais veemente graças a melhora na qualidade de informações e funcionalidades providas ao Grafo PON.

4) É possível validar as materializações criadas em relação aos conceitos implementados?

A resposta para tal questão é positiva. Com base no próprio Grafo PON é possível definir um conjunto de pequenos programas ou, até mesmo, programas completos que podem validar, individualmente ou coletivamente, conceitos do PON em cada materialização proposta. Pertinentemente, as materializações do PON, quando implementadas ou transformadas em *targets* do Sistema de Compilação do PON, herdam implicitamente as características e conceitos já implementados em tal sistema. Isto é, cada *target* já se beneficia de pronto, de boa parte dos conceitos implementados e compartilhados no sistema de compilação, uma vez que as características do PON já estão devidamente

mapeadas em torno do próprio Grafo PON, levando a um maior nível de plenitude de forma natural e implícita neste âmbito.

Esta tese apresenta, em suma, considerável avanço no estado da arte e da técnica do PON, o que toca a área de engenharia de software. Isto se dá, particularmente, por meio do norteamento e quiçá impulsionamento da construção de novas soluções que facilitem o desenvolvimento de software. É sabido que existe uma crescente demanda por software performante, paralelizável e distribuível, conforme contextualizado ao longo deste trabalho.

Nesse âmbito, técnicas e linguagens de programação como as apoiadas pela teoria do PON, sublinhando aqui o método MCPON que se incorpora ao PON, criam caminhos para que tais soluções explorem adequadamente, tanto quanto possível, o paralelismo e a distribuição de qualquer plataforma que os permita. Ainda esses caminhos se dão de maneira simplificada e transparente, justamente por meio de linguagens de programação de alto nível em regime explicitamente declarativo.

Sendo assim, os resultados desta tese tendem a amparar as evoluções subsequentes das tecnologias para o PON, permitindo por consequência colaborar para mitigar, em alguma medida e à luz do tempo futuro, a chamada ‘crise de software’.

7.2 TRABALHOS FUTUROS

Esta seção apresenta as perspectivas de trabalhos futuros relacionados ao tema de pesquisa, as quais orbitam em estabilizar a Tecnologia de Compilação como um todo. Naturalmente, a Tecnologia de Compilação para o PON foi elaborada em regime de prova ou demonstração de conceito, tendo formatos prototipais. Assim sendo, a reformulação da tecnologia proposta à luz de técnicas apropriadas de engenharia de requisitos, engenharia de software, engenharia de testes e afins seria fundamental para estabilizá-la como artefato do estado da técnica. Uma vez que o MCPON tenha um Sistema de Compilação na forma de algo como um produto de software, isso facilitaria criar novas linguagens e novos geradores de código, portanto, constituindo novos compiladores para outros artefatos.

De fato, novas linguagens para o PON e/ou a expansão da atual seriam fundamentais para a tecnologia em si, bem como seu paradigma. Muito embora, tenha

sido possível demonstrar a viabilidade do método MCPON com o Sistema de Compilação na criação da linguagem NOPL como uma linguagem mais completa, espera-se a expansão e maturidade dela. Ainda, espera-se que o método MCPON permita que novas linguagens para o PON sejam criadas se necessário. Tais linguagens poderiam seguir um aspecto mais purista e didático, de modo a instruir novos adeptos ao paradigma, como se apresenta a NOPL ou, por outro lado, seguir um padrão mais direto e pontual, aqui pré-chamada de *NOPL Lite*, especialmente voltadas para especialistas em PON.

Ainda, em termos de compilação, o método prevê que novos *targets* sejam criados e/ou melhorados de forma tal que cada qual se apresente da maneira mais apropriada possível, tanto quanto a plataforma visada permitir, com o objetivo primário de suprir a crescente demanda por software performante, paralelizável e distribuível, conforme contextualiza o Capítulo 1. Os *targets* aqui apresentados já conseguiram demonstrar, em alguma medida, as propriedades elementares do PON e a aplicação de seus conceitos fundamentais em suas essências. Entretanto, compiladores mais maduros, quiçá recompondo assim o estado da técnica, permitiriam demonstrar o PON, com suas propriedades e características, de forma mais veemente, incluso em aplicações industriais ou comerciais.

Certamente, a efetividade de uma materialização para o PON está intimamente ligada à sua construção, reforçando a importância de construções mais industrializadas. O desenvolvedor interessado em construir uma materialização efetiva deveria buscar por técnicas e linguagens que permitam que o *target* visado atinja bons desempenhos e, associado a organização coesa e desacoplada das entidades, busque implementar eficientemente a paralelização na plataforma visada, bem como utilizar de protocolos e técnicas de comunicação em rede para alcançar também a distribuição efetiva.

Por fim, há uma série de outros trabalhos que vão além de estabilizar a tecnologia posta. Esses trabalhos englobam a construção de geradores de códigos multi-plataforma, integração destas tecnologias com a chamada MON (Modelagem Orientada a Notificações) em desenvolvimento por Mendonça (2015) e integração com abordagens de engenharia de sistemas prevista no grupo do PON.

Naturalmente, também todo o arcabouço conceitual, sublinhando o Grafo PON, poderia ser explorado para verificar sua aplicabilidade em outros domínios, incluindo quiçá a tradução de linguagens usuais de outros paradigmas como C/C++,

Java etc para serem executadas sob os princípios do PON e não sob seus paradigmas originais. Isto poderia ser algo particularmente útil como outra contribuição para buscar minimizar a chamada ‘crise do software’ e afins.

REFERÊNCIAS

Este capítulo apresenta as referências bibliográficas utilizadas no presente documento de tese de doutorado. Em tempo, no tocante ao PON, os trabalhos estão listados e não raro disponibilizados em página própria do paradigma. Isso é particularmente pertinente para os Relatórios Técnicos do grupo de pesquisa do PON. Neste âmbito, as páginas pertinentes a esta consideração feita são as que seguem:

http://www.dainf.ct.utfpr.edu.br/~jeansimao/PON/PON.htm
http://www.dainf.ct.utfpr.edu.br/~jeansimao/PON/PON_RelatTecnicos.htm

AHMED, S. **CORBA Programming Unleashed**. Sams Pub., 1998.

AHO, A. V.; LAM, M. S.; SETHI, R.; ULLMAN, J. D. **Compiladores: Princípios, técnicas e ferramentas**. Addison Wesley, 2008.

AI Impacts. **Trends in the cost of computing**. Disponível em: https://aiimpacts.org/trends-in-the-cost-of-computing/. Acesso em: 26 de Março de 2019, Outubro, 2015.

AÏT-KACI, H. **Warren's Abstract Machine, A Tutorial Reconstruction**. Logic Programming Series, MIT Press, 1991.

ALBERT, P. **ILOG Rules, Embedding Rules in C++: Results and Limits**. OOPSLA´94 -Workshop Embedded Object-Oriented Production Systems (EOOPS), 1994.

ASANOVIC, K.; BODIK, R.; CATANZARO, B. C.; GEBIS, J. J.; HUSBANDS, P., EUTZER, K. **The Landscape of Parallel Computing Research : A View from Berkeley**. 2006. Disponível em: <http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf>.

ATHAYDE, E. B; NEGRINI, F. **Implementação de Compilação para C++ Namespaces para a LingPON e Otimizações no Tratamento de Premissas**. Trabalho realizado na disciplina Tópicos Avançados em Engenharia de Software (CAES101 – PPGCA/UTFPR) publicado no Anexo E da Dissertação de Mestrado de Leonardo Araújo Santos. UTFPR. Curitiba, Brasil, 2016.

ATHAYDE, E. B. **Paradigma Orientado a Notificações como Alternativa para Gerenciamento de Energia em Sistemas Embarcados**. Trabalho de Qualificação de Mestrado, , Programa de Pós-Graduação em Computação Aplicada (PPGCA), UTFPR. Curitiba, 2017.

BANASZEWSKI, R. F.; SIMÃO, J. M.; TACLA, P. C.; STADZISZ, P. C. **Notification Oriented Paradigm (NOP) - A Software Development Approach based on Artificial Intelligence Concepts**. VI Congress of Logic Applied to Technology – LAPTEC 2007. Santos, 2007.

BANASZEWSKI, R. F. **Paradigma Orientado a Notificações: Avanços e Comparações**. Dissertação de Mestrado, Pós-Graduação em Engenharia Elétrica e Informática Industrial (CPGEI), UTFPR, Curitiba, 2009.

BANDYOPADHYAY. D.; SEN. J. **Internet of Things: Applications and Challenges in Technology and Standardization**. Wireless Personal Communication, 2011.

BANERJEE, P.; CHANDY, J. A.; GUPTA, M.; HODGES, E. W.; HOLM, J. G.; LAIN, A.; PALERMO, D. J.; RAMASWAMY, S.; SU, E. **The Paradigm Compiler for Distributed Memory Multicomputers**. IEEE Computer 28 (10), pp. 37-47, 1995.

BARRETO, R. M. W. **Notification Oriented Paradigm in the Context of Distributed Systems.** Relatório da disciplina de Tópicos Especiais Em Ec: Paradigma Orientado A Notificações. CPGEI-PPGCA/UTFPR, Curitiba - PR, Brasil, 2016.

BARRETO, W. R. M.; VENDRAMIN, A. C. B. K.; SIMÃO, J. M. **Notification Oriented Paradigm for Distributed Systems**. In: Computer on the Beach 2018, Florianopolis, 2018.

BAUTSCH, M. **Cycles of Software Crises**. ENISA Quarterly on Secure Software, vol. 3, no. 4, p. 3-5, Dez, 2007.

BELMONTE, D.; SIMÃO, J. M.; STADZISZ, P. C. **Proposta de um Método para Distribuição da Carga de Trabalho Usando o Paradigma Orientado a Notificações (PON)**. Revista SODEBRAS, 7(84), p. 10-17, 2012.

BELMONTE, D. **Método para Distribuição da Carga de Trabalho dos Softwares PON em Multicore**. Trabalho de Qualificação de Doutorado, Pós-Graduação em Engenharia Elétrica e Informática Industrial (CPGEI), UTFPR, 2012.

BELMONTE, D.; LINHARES, R. R.; STADZISZ, P. C.; SIMAO, J. M.. **A new Method for Dynamic Balancing of Workload and Scalability in Multicore Systems**. IEEE Latin America Transactions, ISSN: 1548-0992. 2016.

BISON. GNU BISON. Disponível em: https://www.gnu.org/software/bison/. Acesso em: 28 de Abril de 2019.

BORKAR, S.; CHIEN, A. A. **The Future of Microprocessors**. Communications of the ACM, v. 54, n. 5, p. 0–5, 2011.

BROOKSHEAR, G. **Computer Science: An Overview**. Addison Wesley, 2012.

CESARINI, F.; THOMPSON, S. **Erlang Programming: A Concurrent Approach to Software Development**. O'Reilly Media, 1st Edition, 2009.

CHENG, A. M. K; CHEN, J. R. **Response Time Analysis of OPS5 Production Systems**. IEEE Transactions on Knowledge and Data Engineering, vol. 12, n.3, pp. 391-409, 2000.

CODOGNET, P.; DIAZ, D. **wamcc: Compiling Prolog to C**. INRIA-Rocquencourt, Domaine de Voluceau, 1995.

COHEN, J.; HICKEY, T. J. **Parsing and Compiling Using Prolog**. ACM Transactions on Programming Languages and Systems, v. 9, n. 2, p. 125–163, 1987.

COOPER, K.; TORCZON, L. **Engineering: A Compiler**. 2nd Edition, 2011.

COPETTI, L. F. **LINGPON 2.0 & COMPILADOR PARA ASSEMBLY NOCA**. Artigo-Relatório oriundo da disciplina CAES101 do PPGCA/UTFPR, Curitiba – PR, Brasil, 2018.

COULOURIS, G.; DOLLIMORE, J.; KINDBERG, T.; BLAIR, G. **Distributed Systems - Concepts and Designs**. Reading, MA: Addison-Wesley, 2001.

DEDKOV, A. F.; EADLINE, D. J. **Design and Implementation of a Prolog-to-C Compiler**. Paralogic, Inc., 1995.

DEEN, S. M. **Agent-Based Manufacturing: Advances in the Holonic Approach**. ISBN 3-540-44069-0. Springer, 2003.

DIAZ, D.; CODOGNET, P. **Design and Implementation of the GNU Prolog System**. Journal of Functional and Logic Programming, 2001.

DÍAZ, M.; GARRIDO, D.; ROMERO, S.; RUBIO, B.; SOLER, E.; TROYA, J. M. **A component-based nuclear power plant simulator kernel**. Research Articles. Concurrency and Computation: Practice and Experience, 19 (5), pp. 593 - 607, 2007.

DRESCH, A.; LACERDA, D. P.; ANTUNES, J. A. V. J. **Design Science Research**. 1st ed. Porto Alegre: Bookman, 2014.

EKLIND, R. **LLVM IR and GO**. Disponível em: https://blog.gopheracademy.com/advent-2018/lavam-ir-and-go/. Acesso em: 31 de Março de 2019. December, 2018.

ESCHMANN, F.; KLAUER, B.; MOORE, R.; WALDSCHMIDT, K. **SDAARC: An Extended Cache-Only Memory Architecture**. IEEE Micro, 22(3), p. 62-70, 2002.

FAISON, T. **Event-Based Programming: Taking Events to the Limit**. Apress, 2006.

FELDMAN, M. B. **Who's Using Ada?**, SIGAda Education Working Group. Disponível em: http://www2.seas.gwu.edu/~mfeldman/ada-project-summary.html. Acesso em: 29 de Janeiro de 2018. November, 2014.

FERREIRA, C. A. **Linguagem e Compilador para o Paradigma Orientado a Notificações (PON): Avanços e Comparações**. Dissertação de Mestrado, Programa

de Pós-Graduação em Computação Aplicada (PPGCA), UTFPR. Curitiba, Brasil, 2015.

FERREIRA, I. M. **LINGPON 2.0 para o Framework PON C++ 1.0**. Artigo-Relatório oriundo da disciplina CAES101 do PPGCA/UTFPR, Curitiba – PR, Brasil, 2018.

FERSI, G. **Middleware for Internet of Things: a study**. International Conference on Distributed Computing in Sensor Systems. IEEE. p. 230-235, 2015.

FLEX. **The Fast Lexical Analyser**. Disponível em: https://www.gnu.org/software/flex/ Acesso em: 28 de abril de 2019.

FORGY, C. **RETE: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem**. Artificial Intelligence, 19(1), p. 17-37, 1982.

GABBRIELLI, M.; MARTINI, S. **Programming Languages: Principles and Paradigms**. 1st ed. Springer Publishing Company, Incorporated, 2010.

GAUDIOT, J. L.; SOHN, A. S. **Data-Driven Parallel Production Systems**. IEEE Trans. On Software Eng.. V. 16. No 3, pg. 281-293, 1990.

GIARRATANO, J.; RILEY, G. **Expert Systems: Principles and Practice**. Boston, MA: PWS Publishing, 1993.

GNU. **GNU Compiler Collection (GCC) Internals**, Disponível em: https://gcc.gnu.org/onlinedocs/gccint/. Acesso em: 31 de Março de 2019.

GPROLOG. **GNU Prolog**, Disponível em: http://www.gprolog.org/. Acesso em: 31 de Março de 2019.

GRUNE, D.; JACOBS, C. J. H.; REEUWIJK, K. V.; LANGENDOEN, K.; BAL, H. E. **Modern Compiler Design**. Springer, 2012.

GRUVER, W. A. **Distributed Intelligence Systems: A new Paradigm for System Integration**. Proceedings of the IEEE Int. Conference on Information Reuse and Integration (IRI), pg 14-15, 2007.

HANSEN, S.; FOSSUM, T. V. **Event Based Programming**. Kenosha WI, May 23, 2010. Disponível em: http://www.cs.uwp.edu/staff/hansen/EventsWWW/ e http://www.lulu.com/shop/stuart-hansen/event-driven-programming/ebook/product-17346555.html, 2010.

HENZEN, A. F. **Portabilidade do Framework PON de C++ standard para C# e Java.** Relatório da disciplina de Tópicos Especiais Em Ec: Paradigma Orientado A Notificações. CPGEI-PPGCA/UTFPR, Curitiba - PR, Brasil, 2015.

HOPPER, G. M. **The Education of a Computer**, Proceedings of the 1952 ACM National Meeting. Pittsburgh, 1952.

HUGHES, C.; HUGHES, T. **Parallel and Distributed Programming Using C++**. Addison Wesley, 2003.

ILOG. **ILOG Rules for C++ 6.0**. Reference Manual, 1998.

JASINSKI, R. P. **Framework para Geração de Hardware em VHDL a Partir de Modelos em PON (Paradigma Orientado a Notificações)**. Relatório da disciplina de Lógica Reconfigurável por Hardware. Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial (CPGEI), UTFPR, 2012.

JOHNSTON W. M.; PAUL HANNA J. R.; MILLAR, R. J. **Advances in Dataflow Programming Languages**. ACM Computing Surveys, Vol. 36, No. 1, March 2004, pp. 1–34. University of Ulster, 2004.

KAISLER, S. H. **Software paradigms**. John Wiley & Sons, 2005.

KANG, J. A.; CHENG, A. M. K. **Shortening Matching Time in OPS5 Production Systems**. IEEE Transactions on Software Engineering, v. 30, n. 7, p. 448-457, 2004.

KERSCHBAUMER, R. **Paradigma Orientado a Notificações para Síntese de Lógica Reconfigurável**. Qualificação de doutorado – Pós-Graduação em Engenharia Elétrica e Informática Industrial (CPGEI), UTFPR, 2017.

KERSCHBAUMER, R. **Proposição do Paradigma Orientado a Notificações no Desenvolvimento de Circuitos Lógico Digitais Reconfiguráveis**. Tese de doutorado – Pós-Graduação em Engenharia Elétrica e Informática Industrial (CPGEI), UTFPR, 2018.

KERSCHBAUMER, R.; LINHARES, R. R.; SIMÃO, J. M.; STADZISZ, P. C.; LIMA, C. R. E. **Notification Oriented Paradigm to Implement Digital Hardware**. Journal of Circuits Systems and Computers, 2018.

KEYES, R. W. **The Technical Impact of Moore's Law**. IEEE solid state circuits society newsletter, IBM, T. J. Watson Research Center, 2006.

KOSSOSKI, C. **Proposta de um Método de Teste para Processos de Desenvolvimento de Software usando o Paradigma Orientado a Notificações**. Dissertação de Mestrado – Pós-Graduação em Engenharia Elétrica e Informática Industrial (CPGEI), UTFPR, 2015.

KOWALTOWSKI, T. **Von Neumann: suas contribuições à Computação**. Estudos Avançados. vol.10 no. 26. São Paulo Jan./Apr. 1996. Print version ISSN 0103-4014. On-line version ISSN 1806-9592. Brasil, 1996.

KUMAR, V.; LEONARD, N.; MORSE, A. S. **Cooperative Control**. New York: Springer-Verlag, 2005.

LALLY, A; FODOR, P. **Natural Language Processing With Prolog in the IBM Watson System**, Disponível em: https://www.cs.nmsu.edu/ALP/2011/03/natural-

language-processing-with-prolog-in-the-ibm-watson-system/. Acesso em: 29 de Janeiro de 2018. Association for Logic Programming, 2011.

LANES JR, E. A. **LINGPON 2.0 & COMPILADOR PARA FRAMEWORK PON-HD 1.0**. Artigo-Relatório oriundo da disciplina LIN0018 do CPGEI/UTFPR, Curitiba – PR, Brasil, 2018.

LAUTERT, F. **LINGPON 2.0 COM SUPORTE A VETORES**. Artigo-Relatório oriundo da disciplina LIN0018 do CPGEI/UTFPR, Curitiba – PR, Brasil, 2018.

LEE, P.-Y; CHENG, A. M. K. **HAL: a faster match algorithm**. IEEE Transactions on Knowledge and Data Engineering, 14(5), p. 1047-1058, 2002.

LEMONE, K. A. **Fundamentals of Compilers: An Introduction to Computer Language**, Boca Raton: ISBN 0-8493-7341-7, 1992.

LINHARES, R. R.; RONSZCKA, A. F.; VALENÇA, G. Z.; BATISTA, M. V.; WITT F. A.; LIMA, C. R. E.; SIMÃO, J. M.; STADZISZ, P. C. **Comparações entre o Paradigma Orientado a Objetos e o Paradigma Orientado a Notificações sob o contexto de um simulador de sistema telefônico**. COMTEL 2011. Lima, Peru, 2011.

LINHARES, R. R.; SIMÃO, J. M.; STADZISZ, P. C. **Arquitetura de Computador Orientada a Notificações - ARQPON**. Pedido de Patente submetida ao INPI/Brasil (Instituto Nacional de Propriedade Industrial) em 2014. Nº INPI BR1020140040706. Patente submetida ao INPI. Brasil, 2014.

LINHARES, R. R., **Contribuição para o desenvolvimento de uma arquitetura de computação própria ao paradigma orientado a notificações**. Tese de Doutorado, CPGEI, UTFPR. Brasil, 2015.

LINHARES, R. R.; SIMAO, J. M.; STADZISZ, P. C. **NOCA - A Notification-Oriented Computer Architecture**. Latin America Transactions, IEEE (Revista IEEE America Latina), v. 13, n. 5, p. 1593–1604, 2015.

LLVM. **The LLVM Compiler Infrastructure**. Disponível em: https://llvm.org/. Acesso em: 31 de Março de 2019.

LOKE, S. **Context-Aware Pervasive Systems: Architectures for a New Breed of Applications**. 1st Edition, Auerbach Publications (Taylor & Francis Group – USA – 6000 Broken Sound Parkway NW Suite 300 Boca Raton, FL 33487-2742), December 7, 2006, ISBN-10: 0849372550, ISBN-13: 978-0849372551, 1. doi:10.1201/9781420013498, 2006.

LUO L. **GCC processing pipeline**. Disponível em: http://lukeluo.blogspot.com/2014/01/linux-from-scratch-for-cubietruck-c2.html. Acesso em: 31 de Março de 2019. January, 2014.

MARLOW, S. **Parallel and Concurrent Programming in Haskell**. O'Reilly Media, 2013.

MARTINI, G. H. K.; SIMÃO, J. M.; FABRO, J. A. **NOPL & COMPILER TO NOP C++ NAMESPACE MULTI-THREADING ORIENTED SOLUTION: STUDIES OF CAPABILITIES FOR THE X86-64 ARCHITECTURE**. Artigo-Relatório oriundo da disciplina LIN0018 do CPGEI/UTFPR, Curitiba – PR, Brasil, 2018.

MARTINI, G. H. K. **NOP LANGUAGE ON MULTI-CORE ARCHITECTURE COMPUTERS**. Definição Framework NOP AKKA. Aluno Externo CPGEI/UTFPR. Disciplina sobre Paradigma Orientado a Notificações (PON), CPGEI-PPGCA/UTFPR (Profs. J. M. Simão & R. R. Linhares), Curitiba - PR, Brasil, 2018.

MELO, L. C. V.; SIMÃO, J. M.; FABRO, J. A. **Adaptation of the Notification Oriented Paradigm (NOP) for the Development of Fuzzy Systems**. Mathware & Soft Computing, v. 22, n. 1, p. 40–64, 2015.

MELO, L. C., **Adaptação Do Paradigma Orientado A Notificações Para Desenvolvimento De Sistemas Fuzzy**. Dissertação de Mestrado, Programa de Pós-Graduação em Computação Aplicada (PPGCA), UTFPR. Curitiba, Brasil, 2016.

MENDES, C. C. S.; FERREIRA, C. A. **LINGPON 2.0 & COMPILADOR PARA FRAMEWORK PON C++ 3.0 & PON-IP**. Artigo-Relatório independente, mas paralelo a disciplina LIN0018 do CPGEI/UTFPR, Curitiba – PR, Brasil, 2018.

MENDONCA, I. T. M.; SIMÃO, J. M.; WIECHETECK, L. V. B.; STADZISZ, P. C. **Método para Desenvolvimento de Sistemas Orientados a Regras utilizando o Paradigma Orientado a Notificações**. In: LA-CCI/CBIC, 2015, Curitiba. LA-CCI/CBIC, 2015.

MENDONÇA, I. T. M., **Metodologia De Projeto De Software Orientado A Notificações**. Trabalho de Qualificação de Doutorado doutorado – Pós-Graduação em Engenharia Elétrica e Informática Industrial (CPGEI), UTFPR, 2016.

MIRANKER, D. P. **TREAT: A better Match Algorithm for AI Production Systems**. Sixth National Conference on Artificial Intelligence - AAAI'87, pp. 42-47, 1987.

MIRANKER, D. P.; BRANT, D. A.; LOFASO, B.; GADBOIS, D. **On the Performance of Lazy Matching in Production Systems**. 8th National Conference on Artificial Intelligence AAAI (pp. 685-692). AAAI Press / The MIT Press, 1990.

MIRANKER, D. P.; LOFASO, B. **The organization and Performance of a TREAT-based Production System Compiler**. IEEE Trans. on Knowledge and Data Engineering, III (1), p. 3-10, 1991.

MOORE, G. E. **Cramming More Components Onto Integrated Circuits**. Electronics Magazine, 1965.

NEGRINI, F. **LINGPON 2.0 & COMPILADOR PARA FRAMEWORK ERLANG: PROGRAMAÇÃO MULTICORE TRANSPARENTE JÁ É UMA REALIDADE**. Artigo-Relatório independente, mas tecnicamente paralelo a disciplina LIN0018, depois elaborado para a disciplina MC0001, ambas do CPGEI/UTFPR, Curitiba – PR, Brasil, 2019.

NOVAES, P. J. D.; SIMÃO, J. M.; STADZISZ, P. C. **Integration between Requirements Modeling and Software Development in the Notification Oriented Paradigm: A Security System Case Study**. In: Computer on the Beach 2018, Florianópolis – SC, 2018.

OLIVEIRA, S.; STEWART, D. E. **Writing Scientific Software: A Guided to Good Style**. Cambridge University Press, 2006.

OLIVEIRA, R. N. **Uso do Paradigma Orientado a Notificações em Sistemas Sencientes.** Relatório da disciplina de Tópicos Especiais Em Ec: Paradigma Orientado A Notificações. CPGEI-PPGCA/UTFPR, Curitiba - PR, Brasil, 2016.

OLIVEIRA, R. N.; ROTH, V.; HENZEN, A. F.; SIMÃO, J. M.; WILLE, E. C. G.; NOHAMA, P. **Notification Oriented Paradigm Applied to Ambient Assisted Living Tool**. IEEE Latin America Transactions, 2018.

OSHIRO, L. K. **LINGPON 2.0 & COMPILADOR PARA SOLUÇÃO PON EM C++ ORIENTADO A ESPAÇO DE NOMES**. Artigo-Relatório oriundo da disciplina LIN0018 do CPGEI/UTFPR, Curitiba – PR, Brasil, 2018.

PAAKI, J. **Prolog in Practical Compiler Writing**. In The Computer Journal, v. 34, n. 1, 1991.

PAN, J.; SOUZA, G. N.; KAK; A. C. **FuzzyShell: A large-scale expert system shell using fuzzy logic for uncertainty reasoning**. IEEE Trans. Fuzzy Syst., vol. 6, no. 4, pp. 563–581, Nov. 1998.

PETERS, E. **Coprocessador para Aceleração de Aplicações Desenvolvidas Utilizando Paradigma Orientado a Notificações**. 2012. Dissertação de Mestrado – Pós-Graduação em Engenharia Elétrica e Informática Industrial (CPGEI), UTFPR, 2012.

PETERS, E.; JASINSKI, R. P.; PEDRONI, V. A.; SIMAO, J. M. **A new hardware coprocessor for accelerating Notification-Oriented applications**. International Conference on FieldProgrammable Technology (FPT), South Korea, 2012.

PIERCE, B. C. **Types and Programming Languages**. MIT Press, 2002.

PIMENTEL, A. R.; STADZISZ, P. C. **Application of the Independence Axiom on the Design of Object-Oriented Software Using the Axiomatic Design Theory**. Journal of Integrated Design & Process Science, v. 10, 2006.

PON. **Relatórios Técnicos do PON**. Acesso em: 18 de Abril de 2019. Disponível em: http://www.dainf.ct.utfpr.edu.br/~jeansimao/PON/PON_RelatTecnicos.htm.

PORDEUS, L. F. **Notification-Oriented Paradigm (NOP): CTA Simulator**. Disciplina sobre Paradigma Orientado a Notificações (PON) – CPGEI-PPGCA/UTFPR, 2015.

PORDEUS, L. F.; KERSCHBAUMER, R.; LINHARES, R. R.; WITT, F. A.; STADZISZ, P. C.; LIMA, C. R. E.; SIMÃO, J. M. **Notification Oriented Paradigm to Digital Hardware**. Revista SODEBRAS, v. 11, p. 116-122, 2016.

PORDEUS, L. F. **Simulação de uma Arquitetura de Computação Própria ao Paradigma Orientado a Notificações**. Dissertação de Mestrado – Pós-Graduação em Engenharia Elétrica e Informática Industrial (CPGEI), UTFPR, 2017.

PROKOPEC, A. **Learning Concurrent Programming in Scala**. Packt Publishing, 2014.

RAYMOND, E. S. R. **The Art of UNIX Programming**. Pp. 327, A. Wesley, 2003.

REILLY; D. R.; REILLY, M. **Java Network Programming and Distributed Computing**. Addison-Wesley, 2002.

RENAUX, P. B. **NOPL & Compiler to NOP C# FRAMEWORK**. Artigo-Relatório oriundo da disciplina CAES101 do PPGCA/UTFPR, Curitiba – PR, Brasil, 2018.

RICARTE, I. **Introdução à compilação**. Editora Campos, 2008.

RONSZCKA, A. F. **Contribuição para a Concepção de Aplicações no Paradigma Orientado a Notificações (PON) sob o viés de Padrões**. Dissertação de mestrado – Pós-Graduação em Engenharia Elétrica e Informática Industrial (CPGEI), UTFPR, 2012.

RONSZCKA, A. F.; FERREIRA, C. A.; IORIS, P. A. M.; KOSSOSKI, C. **Compilador para o Paradigma Orientado a Notificações**. Trabalho realizado nas disciplinas Tópicos Especiais em EC: Linguagens e Compiladores (TEC0302 – CPGEI/UTFPR) e Tópicos Avançados em Engenharia de Software (CAES101 – PPGCA/UTFPR) publicado no Apêndice A da Dissertação de Mestrado de Cleverson Avelino Ferreira. UTFPR. Curitiba, Brasil, 2013.

RONSZCKA, A. F.; BANASZEWSKI, R. F.; LINHARES, R. R.; TACLA, C. A.; STADZISZ, P. C.; SIMAO, J. M. **Notification-Oriented and Rete Network Inference: A Comparative Study**. Systems, Man, and Cybernetics (SMC), 2015 IEEE International Conference on. p. 807–814, 2015.

RONSZCKA, A. F.; VALENÇA, G. Z.; LINHARES, R. R.; FABRO, J. A.; STADZISZ, P. C.; SIMÃO, J. M. **Notification-Oriented Paradigm Framework 2.0: An Implementation Based On Design Patterns**. IEEE Latin Americca Transactions – v. 15. p. 2220-2231, 2017.

RONSZCKA, A. F.; FERREIRA, C. A.; STADZISZ, P. C.; FABRO, J. A.; SIMÃO, J. M. **Notification-Oriented Programming Language and Compiler**. SBESC – VII Brazilian Symposium on Computing Systems Engineering, 2017.

RONSZCKA, A. F. **LingPON – Linguagem de Programação e Compilador para o Paradigma Orientado a Notificações (PON) – Uma Materialização Efetiva para a Validação das Propriedades Elementares do PON**. Trabalho de Qualificação de

doutorado – Pós-Graduação em Engenharia Elétrica e Informática Industrial (CPGEI), UTFPR, 2018.

ROSEN, B; WEGMAN, M. N.; ZADECK, F. K. **Global value numbers and redundant computations**. Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1988.

ROY, P. VAN; HARIDI, S. **Concepts, Techniques, and Models of Computer Programming**. Cambridge, MA, USA: MIT Press, 2004.

ROY, P. VAN. **Programming Paradigms for Dummies: What Every Programmer Should Know**. New Computational Paradigms for Computer Music, p. 9–47, 2009.

RUSSEL, S. J.; NORVIG, P. **Inteligência Artificial**. ed. Editora Campus, 2003.

RUSSINOFF, D. M. **A verified prolog compiler for the Warren Abstract Machine**. In Journal of Logic Programming, v. 13, n. 4, p. 367–412, 1992.

SANTOS, L. A. **Linguagem e Compilador para o Paradigma Orientado a Notificações: avanços para facilitar a codificação e sua validação em uma aplicação de controle de futebol de robôs**. Dissertação de Mestrado – Pós-Graduação em Engenharia Elétrica e Informática Industrial (CPGEI), UTFPR, 2017.

SCHÜTZ, F.; SANTOS, L. A.; PORDEUS, L. F.; KERSCHBAUER, R. **Manual da LingPON: Evoluções na linguagem e compilador**. Trabalho realizado nas disciplinas Tópicos Especiais em EC: Linguagens e Compiladores (TEC0302 – CPGEI/UTFPR) e Tópicos Avançados em Engenharia de Software (CAES101 – PPGCA/UTFPR) publicado no Anexo D da Dissertação de Mestrado de Leonardo Araújo Santos. UTFPR. Curitiba, Brasil, 2015.

SCHÜTZ, F.; Fabro J. A.; RONSZCKA, A. F.; STADZISZ, P. C.; SIMÃO, J. M. **Proposal of a declarative and parallelizable artificial neural network using the notification-oriented paradigm**. Neural Computing and Applications, p. 1-12, 2018.

SCHÜTZ, F. **Neuro-PON: Uma abordagem para o Desenvolvimento de Redes Neurais Artificiais utilizando o Paradigma Orientado a Notificações**. Tese de doutorado – Pós-Graduação em Engenharia Elétrica e Informática Industrial (CPGEI), UTFPR, 2019.

SCOTT, M. L. **Programming Language Pragmatics**. Ed. 4. Elsevier Science & Technology., 2016.

SCRIPTOL. **List of Hello World Programs in 200 Programming Languages**. Disponível em: http://www.scriptol.com/programming/hello-world.php. Acesso em: 27 de Janeiro de 2018.

SEBESTA, R. W. **Concepts of Programming Languages**. 10th edition. Pearson, 2012.

SEVILLA, D.; GARCIA, J. M.; GÓMEZ, A. **Using AOP to Automatically Provide Distribution, Fault Tolerance, and Load Balancing to the CORBA-LC Component Model**. Parallel Computing: Architectures, Algorithms and Applications, C. Bischof, M. Bücker, P. Gibbon, G. R. Joubert, T. Lippert, B. Mohr, F. Peters (Eds.), John von Neumann Institute for Computing, Jülich, NIC Series, Vol. 38, ISBN 978-3-9810843-4-4, pp. 347-354, 2007. Reprinted in: Advances in Parallel Computing, Volume 15, ISSN 0927-5452, ISBN 978-1-58603-796-3 (IOS Press), 2008.

SILIO, L. T. **LINGPON 2.0 & COMPILADOR PARA FRAMEWORK JAVA 1.0**. Artigo-Relatório oriundo da disciplina CAES101 do PPGCA/UTFPR, Curitiba – PR, Brasil, 2018.

SIMÃO, J. M. **Proposta de uma Arquitetura de Controle para Sistemas Flexíveis de Manufatura Baseada em Regras e Agentes**. 2001. Dissertação de Mestrado, Universidade Tecnológica Federal do Paraná - UTFPR, Curso de Pós-Graduação em Engenharia Elétrica e Informática Industrial - CPGEI, Curitiba, 2001.

SIMÃO, J. M.; STADZISZ, P. C. **An Agent-Oriented Inference Engine applied for Supervisory Control of Automated Manufacturing Systems**. In: Abe J. M., Silva Filho J. I.. (Org.). Frontiers in Artificial Intelligence and Applications (Advances in Logic, Artificial Intelligence and Robotics ? LAPTEC 2002). Amsterdam, The Netherlands: IOS PRESS BOOKS, 2002, v. 85, p. 234-241, 2002.

SIMÃO, J. M.; FABRO, J. A.; STADZISZ, P. C. **An Agent-Oriented Fuzzy Inference Engine**. In: 6. Simpósio Brasileiro de Automação Inteligente, 2003, Bauru-SP. Anais do VI SBAI, 2003.

SIMÃO, J. M. **A Contribution To The Development Of A HMS Simulation Tool And Proposition Of A Meta-Model For Holonic Control**. 2005. Tese de doutorado. CPGEI, CEFET-PR. Curitiba, Brasil, 2005.

SIMÃO, J. M.; STADZISZ, P. C. **Paradigma Orientado a Notificações (PON) – Uma Técnica de Composição e Execução de Software Orientado a Notificações**. Pedido de Patente submetida ao INPI/Brasil (Instituto Nacional de Propriedade Industrial) em 2008 e a Agência de Inovação/UTFPR em 2007. Nº INPI PI08055181. Patente submetida ao INPI. Brasil, 2008.

SIMÃO, J. M.; STADZISZ, P. C. **Inference Based on Notifications: A Holonic Metamodel Applied to Control Issues**. IEEE Transactions on Systems, Man and Cybernetics. Part A, Systems and Humans, v. 39, p. 238-250, 2009.

SIMÃO, J. M., TACLA, C. A., STADZISZ, P. C., **Holonic Control Meta-Model**. IEEE Transactions on Systems, Man and Cybernetics. Part A, Systems and Humans, 2009.

SIMÃO, J. M.; STADZISZ, P. C. **Mecanismo de Resolução de Conflito e Garantia de Determinismo para o Paradigma Orientado a Notificações (PON)**. Pedido de Patente submetida ao INPI/Brasil (Instituto Nacional de Propriedade Industrial) em 2010. Nº INPI PI10002960. Brasil, 2010.

SIMÃO, J. M.; BANASZEWSKI, R. F; TACLA, C. A.; STADZISZ, P. C. **Mecanismo de Inferência Otimizado do Paradigma Orientado a Notificações (PON) e Mecanismos de Resolução de Conflitos para Ambientes Monoprocessados e Multiprocessados Aplicados ao PON**. Pedido de Patente submetida ao INPI/Brasil (Instituto Nacional de Propriedade Industrial) em 2010. Nº INPI PI10037365. Brasil, 2010.

SIMÃO, J. M.; BANASZEWSKI, R. F.; TACLA, C. A.; STADZISZ, P. C. **Notification Oriented Paradigm (NOP) and Imperative Paradigm: A Comparative Study**. Journal of Software Engineering and Applications (JSEA), 2012.

SIMÃO, J. M.; BELMONTE, D. L.; RONSZCKA, A. F.; LINHARES, R. R.; VALENÇA, G. Z.; BANASZEWSKI, R. F.; FABRO, J. A.; TACLA, C. A.; STADZISZ, P. C.; BATISTA, M. V. **Notification Oriented and Object Oriented Paradigm Comparison via Sale System**. Journal of Software Engineering and Applications (JSEA), 2012.

SIMÃO, J. M.; BELMONTE, D. L.; VALENÇA, G. Z.; BATISTA, M. V.; LINHARES, R. R.; BANASZEWSKI, R. F.; FABRO, J. A.; TACLA, C. A.; STADZISZ, P. C.; RONSZCKA, A. F. **A Game Comparative Study: Object-Oriented Paradigm and Notification-Oriented Paradigm**. Journal of Software Engineering and Applications (JSEA), 2012.

SIMÃO, J. M.; STADZISZ, P. C.; WIECHETECK, L. V. B. **Perfil UML para o Paradigma Orientado a Notificações (PON), Perfil UML para o Paradigma Orientado a Regras (POR), Método de Desenvolvimento Orientado a Notificações (DON) e Método de Desenvolvimento Orientado a Regras (DOR)**. Pedido de Patente submetida ao INPI/Brasil (Instituto Nacional de Propriedade Industrial) em 2012. Nº INPI BR1020120264307. Patente submetida ao INPI. Brasil, 2012.

SIMÃO, J. M.; LINHARES, R. R.; WITT, F. A.; LIMA, C. R. E.; STADZISZ, P. C. **Paradigma Orientado a Notificações em Hardware Digital**. Pedido de Patente submetida ao INPI/Brasil (Instituto Nacional de Propriedade Industrial) em 2012. Nº INPI BR102012026429. Patente submetida ao INPI. Brasil, 2012.

SIMÃO, J. M.; RENAUX, D. P. B.; LINHARES, R. R.; STADZISZ, P. C. **Evaluation of the Notification Oriented Paradigm applied to Sentient Computing**. In: 10th Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS 2014) in 2014 IEEE 17th International Symposium on Object/Component-Oriented Real-Time Distributed Computing, Reno - Nevada – USA, v. 1555-0. p. 253-260, 2014.

SIMÃO, J. M.; PANETTO, H.; LIAO, Y.; STADZISZ, P. **A Notification-Oriented Approach for Systems Requirements Egineering**. In: 23rd ISPE International Conference on Transdisciplinary Engineering. Advances in Transdisciplinary Engineering. Amsterdam: IIOS Press, 2016. v. 4. p. 229-238, 2016.

SIMÃO, J. M.; STADZISZ, P. C. **Framework Referencial PON C++**. Patente de Programa de Computador no INPI, Número do registro: BR5120170010-1, data de registro: 03/08/2017, Curitiba - PR, Brasil, 2017.

SIMÃO, J. M.; BANASZEWSKI, R. F.; TACLA, C. A.; STADZISZ, P. C. **Framework PON C++ 1.0**. Patente de Programa de Computador no INPI, Número do registro: BR512017001015-3, data de registro: 03/08/2017, Curitiba - PR, Brasil, 2017.

SIMÃO, J. M.; RONSZCKA, A. F.; VALENÇA, G. Z.; LINHARES, R. R.; STADZISZ, P. C. **Framework PON C++ 2.0**. Patente de Programa de Computador no INPI, Número do registro: BR51201700149-5, data de registro: 01/12/2017, Curitiba - PR, Brasil, 2017.

SIMÃO, J. M. **Consolidação do Paradigma Orientado a Notificações (PON)**. 2018. Chamada Pública 15/2017. Programa de Bolsas de Produtividade em Pesquisa e Desenvolvimento Tecnológico / Extensão. Projeto Aprovado. Fundação Araucária, 2018.

SOUZA, J. T. S. de; FABRO J. A.; BANASZEWSKI, R. F.; SIMÃO, J. M. **Proposta de uma Máquina de Inferência Fuzzy utilizando o Paradigma Orientado a Notificações (PON)**. In: IV Congresso da Academia Trinacional de Ciências - 2009, Foz do Iguaçu, 2009.

SPIVAK, R. **Let's Build A Simple Interpreter**. Disponível em: https://ruslanspivak.com/lsbasi-part7/. Acesso em: 18 de Abril de 2019. December, 2015.

TALAU, M. **PONIP: Uso do Paradigma Orientado a Notificações em Redes IP**. Relatório da disciplina de Tópicos Especiais Em Ec: Paradigma Orientado A Notificações. CPGEI-PPGCA/UTFPR, Curitiba - PR, Brasil, 2016.

TANENBAUM, A. S.; STEEN, M. **Distributed Systems: Principles and Paradigms**, Prentice-HALL, 2002.

TIANFIELD, H. T. **A New Framework of Holonic Self-organization for Multi-Agent Systems**. IEEE Int. Conf. on System, Man & Cyb., 2007.

TILEVICH, E.; SMARAGDAKIS, Y. **J-Orchestra: Automatic Java Application Partitioning**. 16th European Conf. on Object-Oriented Programming, pg 178-204, B. Magnusson (Ed), Springer, 2002.

TIOBE. **TIOBE Index for April 2019**. Disponível em: https://www.tiobe.com/tiobe-index/. Acesso em: 14 de Abril de 2019.

TUCKER, A. B.; NOONAN, R. E. **Linguagens de Programação: Princípios e Paradigmas**. AMGH Editora Ltda., 2010.

TUTTLE, S. M.; EICK, C. F. **Suggesting Causes of Faults in Data-Driven Rule-Based Systems**. Proc. of the IEEE 4th International Conference on Tools with Artificial Intelligence, pg 413-416, Arlington, VA., 1992.

VALENÇA, G. Z. **Contribuição para Materialização do Paradigma Orientado a Notificações (PON) Via Framework e Wizard**. Dissertação de Mestrado, Programa

de Pós-Graduação em Computação Aplicada (PPGCA), UTFPR. Curitiba, Brasil, 2012.

WACHTER, B.; MASSART, T.; MEUTER, C. **dSL: An Environment with Automatic Code Distribution for Industrial Control Systems**. Proc. of the 7th Int. Conf. on Principles of Distributed Syst., 2003, La Martinique, France, V. 3144 of LNCS, pg 132-45, Springer, 2004.

WARREN, D. H. D. **Logic Programming and Compiler Writing**. In Software - Practice and Experience, v. 10, p. 97–125, 1980.

WARREN, D. H. D. **An Abstract Prolog Instruction Set**. Technical Note 309 in Artificial Intelligence Center at SRI Project, 1983.

WATT, D. A. **Programming Language Design Concepts**. J. Willey & Sons, 2004.

WEBER, L.; BELMONTE, D. L.; BANASZEWSKI, R. F.; STADZISZ, P. C.; SIMÃO, J. M. **Viabilidade de Controle Orientado a Notificações (CON) em Ambiente Concorrente Baseado em Threads**. In: Seminário de Iniciação Científica e Tecnológica (SICITE 2010), 2010, Cornélio Procópio. SICITE'2010, 2010.

WEXELBLAT, R. L. **History of Programming Languages**, New York: Academic Press. ISBN 0-12-745040-8, 1981.

WIECHETECK, L. V. B. **Método para projeto de software usando o paradigma orientado a notificações – PON**. Dissertação de Mestrado – Pós-Graduação em Engenharia Elétrica e Informática Industrial (CPGEI), UTFPR, 2011.

WIECHETECK, L. V. B.; STADZISZ, P. C.; SIMÃO, J. M. **Um Perfil UML para o Paradigma Orientado a Notificações (PON)**. COMTEL 2011. Lima, Peru, 2011.

WITT, F. A.; SIMAO, J. M.; LINHARES, R. R.; STADZISZ, P. C.; LIMA, C. R. E. **Comparação entre o Paradigma Orientado a Objetos (POO) e o Paradigma Orientado a Notificações (PON) em um Controle Discreto em Lógica Reconfigurável**. Em: XVI SICITE - Seminário de Iniciação Científica e Tecnológica da UTFPR, 2011, Ponta Grossa - PR. Anais do XVI SICITE, 2011.

XAVIER, R. D. **Paradigmas de Desenvolvimento de Software: Comparação entre abordagens Orientada a Eventos e Orientada a Notificações**, Dissertação de Mestrado, Programa de Pós-Graduação em Computação Aplicada (PPGCA), UTFPR. Curitiba, Brasil, 2014.

APÊNDICE A

LINGUAGENS DE PROGRAMAÇÃO

De maneira a apresentar as características e particularidades das linguagens de programação, a Seção A.1 apresenta uma breve história das linguagens de programação e, portanto, de seus paradigmas, bem como apresenta as principais linguagens utilizadas nos dias atuais. A Seção A.2, por sua vez, apresenta algumas características para a avaliação de linguagens de programação em relação a sua possível popularidade.

A.1 História das linguagens de programação

Esta seção em particular descreve o surgimento de algumas das principais linguagens de programação na história da computação. Ademais, a seção explora o ambiente em que cada qual foi desenvolvida e as motivações por trás de seu desenvolvimento. É importante ressaltar que esta seção é uma adaptação do texto apresentado no livro de Sebesta (2012), no qual é possível encontrar a história de cada uma das linguagens citadas em maiores detalhes.

De maneira geral, a história começa entre os anos de 1936 e 1945, na qual o cientista alemão Konrad Zuse (pronuncia-se “Tzu-ze”) construiu uma série de computadores complexos e sofisticados a partir de relés eletromecânicos. Mais adiante, em 1945, no fim da segunda guerra mundial, a maioria destes foram destruídos, a não ser um de seus últimos modelos, o Z4. Então, Zuse se mudou para uma remota aldeia bávara chamada de Hinterstein, enquanto os membros de seu grupo de pesquisas seguiram caminhos distintos.

Trabalhando sozinho, Zuse embarcou em um esforço para desenvolver uma linguagem para expressar computações. Tal projeto foi iniciado em 1943 como proposta para sua tese de Ph.D. Ele chamou sua linguagem de Plankalkül que significa cálculo de programa. A Plankalkül era notavelmente complexa, com alguns de seus recursos mais avançados na área de estruturas de dados. A linguagem era baseada em um sistema formal de notação para algoritmos, a qual poderia lidar apenas com planos de cálculos lineares, sem ramificações (*if-then*) e sem laços de repetição (*loops*).

Em um manuscrito extenso datado de 1945, mas não-publicado até 1972, ele definiu a Plankalkül e escreveu algoritmos na linguagem para uma ampla variedade de problemas. O manuscrito de Zuse continha programas de complexidade bem maior do que qualquer um escrito antes de 1945. Inclusos, havia programas para classificar arranjos de números, para testar a conectividade de determinado grafo e para executar operações com números inteiros e reais, inclusive raiz quadrada. Talvez o mais notável tenha sido suas 49 páginas de algoritmos para jogar xadrez, um jogo em que ele não era especialista. É importante ressaltar que por estar em uma época conturbada, em plena guerra e em isolamento na Alemanha, o trabalho de Zuse poderia ter tomado outras proporções se tivesse sido conhecido na época.

Paralelamente aos esforços de Zuze, na Inglaterra, outros cientistas, trabalhavam em projetos normalmente secretos e sigilosos (dada a época conturbada), como o caso de Alan Turing e seus colaboradores que entre os anos de 1941 e 1945 trabalharam no desenvolvimento de máquinas que ficaram conhecidas como *Bombs* e *Colossus*, dedicadas à criptoanálise. Entre os anos de 1942 e 1945, outro projeto, este alocado nos Estados Unidos e liderado por Eckert e Mauchly, foi a criação do primeiro computador de propósito geral completamente eletrônico, chamado de ENIAC.

Mais tarde, John Von Neumann se juntou ao projeto ENIAC e participou de um grupo para a concepção de um computador de programa armazenado, o EDVAC. Neste, as instruções deveriam ser armazenadas na memória do computador. Até então elas eram lidas de cartões perfurados e executadas, uma a uma. Armazená-las na memória, para então executá-las, tornaria o computador mais rápido, já que no momento da execução, as instruções seriam obtidas com rapidez eletrônica. Como parte desse grupo, ele se ofereceu para escrever uma descrição do mesmo, denominada de arquitetura de von Neumann (KOWALTOWSKI, 1996).

Nesse âmbito, existe controvérsia quanto a quem teria sido o primeiro a propor o conceito de programa armazenado em memória. O trabalho teórico de Turing, datado em 1936, com o qual von Neumann estava familiarizado, já indicava essa possibilidade. Por outro lado, existem algumas referências ao assunto, bastante obscuras e ambíguas, em algumas fontes anteriores ao documento produzido por von Neumann, além das afirmações posteriores de Eckert, Mauchly e outros. Não há dúvida de que a ideia de programa armazenado estava no ar e é bastante provável

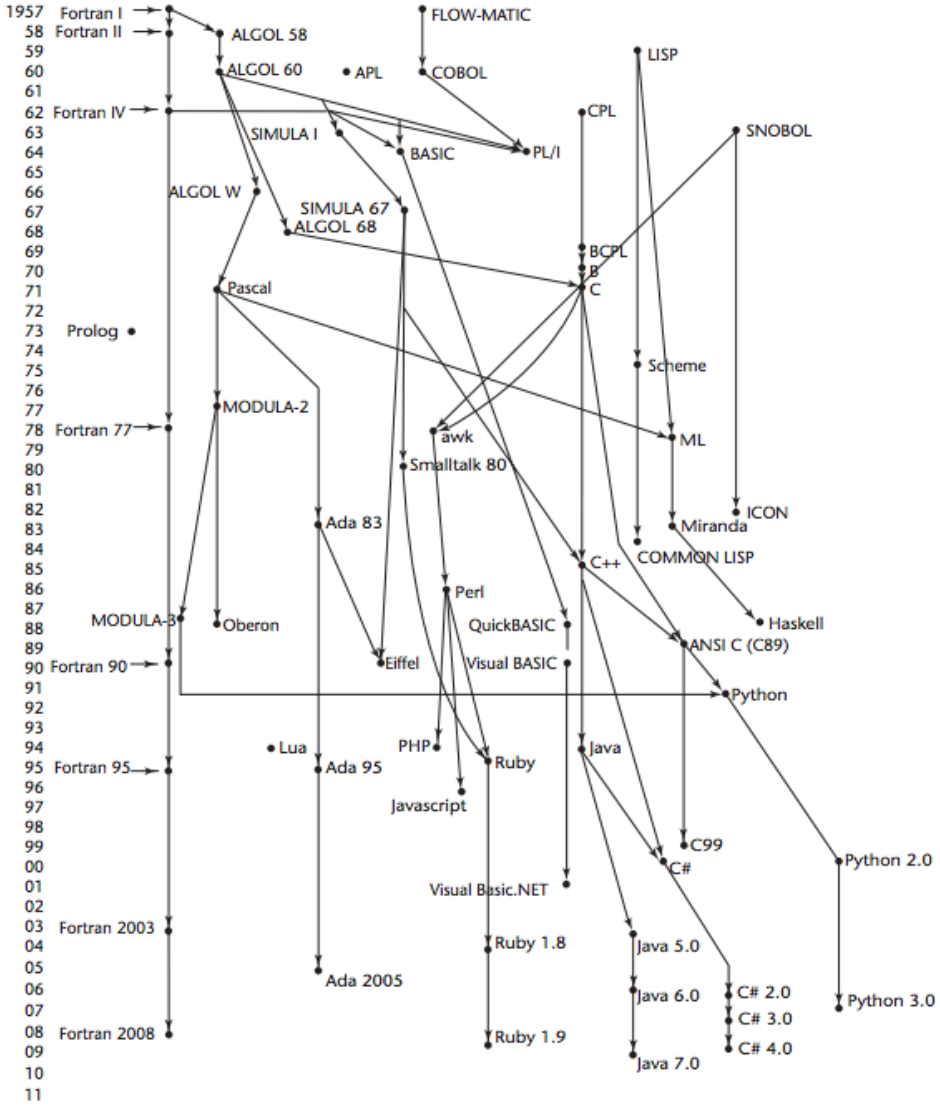
que tenha sido sugerida por mais de uma pessoa ou nascido no meio de discussões sobre o projeto do EDVAC (KOWALTOWSKI, 1996).

Independentemente de quem tenha sido primeiro a sugerir a ideia de programa armazenado na memória, o fato é que o documento redigido por von Neumann é a primeira descrição minuciosa e quase completa da arquitetura de um computador desse tipo, com repertório de operações que permite a utilização plena dos seus recursos. Dado esse contexto histórico, a arquitetura Von Neumann foi a que prevaleceu.

Em geral, os computadores que se tornaram disponíveis no final da década de 40 e no início da década de 50 eram bem menos práticos do que os de hoje. Além de serem lentos, pouco confiáveis e caros, assim como baseados em memórias com pouca capacidade, tais máquinas eram difíceis de programar, principalmente por causa da falta de software de apoio. Não havia nenhuma linguagem de programação de alto nível ou mesmo linguagens de montagem. A programação, naquela época, era feita em código de máquina, o que, além de tedioso, era propenso a erros.

Com esse problema então posto naquela época, milhares de linguagens surgiram daquele momento em diante na história da computação. Entretanto, apenas uma fração bem menor destas tiveram algum sucesso. De modo a ilustrar a genealogia das linguagens de programação em alto nível mais comuns da história da computação, a Figura 79 apresenta-as em ordem cronológica.

Figura 79: Genealogia das linguagens mais comuns da história da computação



Fonte: Sebesta, 2012

Conforme ilustra a Figura 79, a história das linguagens tem suas origens em meados de 1957. Certamente, um dos avanços mais significativos na história da computação surgiu com o advento do IBM 704 em 1954. Grande parte desse avanço foi a motivação para a criação da famosa linguagem de programação FORTRAN[24]. A IBM foi a primeira grande corporação a ter os recursos para o desenvolvimento de tais tecnologias naquela época.

O hardware era bastante restrito, o armazenamento acontecia em fitas magnéticas, as memórias eram de núcleo de ferrite e os circuitos transistorizados.

[24] Linguagem suportada pelos (sub)paradigmas estruturado, imperativo, procedural e orientado a objetos.

Muitos programadores, naquela época, programavam à mão diretamente em linguagem de máquina, devido a eficiência de execução em comparação a alternativa precursora das linguagens de alto nível, os chamados sistemas pseudocódigo interpretativos.

A linguagem de programação FORTRAN foi considerada por muitos como a primeira linguagem de programação imperativa a ser criada na história da computação, além de estar em plena utilização nos dias atuais[25]. Em linhas gerais, ela foi desenvolvida a partir de um projeto da IBM, o qual foi liderado por John Backus. Em sua primeira publicação, referido nos dias de hoje como FORTRAN 0, antes de sua implementação, a linguagem foi promovida como a solução para os problemas dos programadores da época, garantindo a eficiência dos programas codificados a mão e a facilidade de programação dos sistemas pseudocódigo interpretativos. Além disso, seu primeiro compilador incluía inclusive um verificador de erros de sintaxe. Essa versão preliminar foi modificada e prosseguiu até o lançamento oficial em 1957, surgindo então a chamada FORTRAN I.

Em paralelo a abordagem imperativa adotada pela linguagem FORTRAN, naquela mesma época houve um crescente interesse em Inteligência Artificial. Em 1958, John McCarthy, do MIT, assumiu um cargo de verão no *Information Research Department* da IBM. Sua meta era investigar as computações simbólicas e como problema experimental, escolheu a diferenciação de expressões algébricas. A partir desse estudo surgiu uma lista de requisitos, entre eles estavam os métodos de fluxo de controle de funções matemáticas como recursão e expressões condicionais.

A única linguagem de alto nível, na época, o FORTRAN, não tinha nenhuma dessas funções aventadas. Nesse âmbito, o primeiro esforço importante foi produzir um sistema para processamento de listas. Tal aplicação impulsionou o desenvolvimento da linguagem de processamento de listas, o LISP, criando conjuntamente o paradigma hoje conhecido como funcional. A partir do LISP surgiram algumas linguagens de programação que seguem o paradigma funcional, tais como Scheme, Common Lisp, ML, Miranda, Haskell e Scala[26]. Muitas delas ainda são

---

[25] De acordo com o TIOBE Index (2019), a linguagem Fortran está em 25º lugar na lista das linguagens mais utilizadas atualmente.

[26] Todas as linguagens citadas suportam o paradigma funcional, porém LISP, Scheme, Common Lisp, ML e Scala também suportam o estilo imperativo. Por outro lado, Miranda e Haskell suportam um estilo puramente funcional, com avaliação preguiçosa (*lazy*), que basicamente retarda a

bastante utilizadas atualmente e tem estudos principalmente voltados a programação concorrente[27].

Ainda, sob influência do FORTRAN, outra linguagem que merece destaque é a linguagem de programação ALGOL, baseada no paradigma imperativo-procedimental que surgiu com a natureza imperativa dos códigos de máquina. Em suma, a linguagem de programação ALGOL passou a existir como resultado dos esforços de um comitê de programadores influentes da época para projetar uma linguagem universal. Em 1957, quando o FORTRAN havia se tornado uma realidade, diversas outras linguagens imperativas de alto nível estavam sendo desenvolvidas.

Essa proliferação de linguagens tornou difícil a comunicação entre utilizadores delas. Além disso, novas linguagens desenvolviam-se em torno de arquiteturas únicas, algumas para computadores UNIVAC e outras para as máquinas da IBM, todas variações da arquitetura geral de von Neumann. Em resposta a essa proliferação, diversos grupos de grandes utilizadores de computadores submeteram uma petição para formarem um comitê de estudos e para recomendar ações voltadas à criação de uma linguagem universal.

Ainda que o FORTRAN pudesse ter sido candidato, ele não poderia se tornar universal, porque na época era propriedade somente da IBM. Com base nas colaborações e definições do comitê, a nova linguagem chamada de ALGOL deveria possuir as seguintes características:

- A sintaxe da linguagem deveria estar o mais próximo possível da notação matemática padrão e os programas, nela escritos, deveriam ser legíveis e com pouca explicação adicional;
- Deveria ser possível usar a linguagem para a descrição de processos computacionais em publicações;
- Os programas na nova linguagem deveriam ser mecanicamente traduzíveis para linguagem de máquina.

---

avaliação de uma expressão até que seu valor seja necessário (avaliação não rigorosa) e que também evita avaliações repetidas (compartilhamento).

[27] De acordo com o TIOBE Index (2019), as linguagens do paradigma funcional são bastante utilizadas atualmente. Na lista, Scala está no 31º lugar, LISP está em 33º, Scheme em 38º, Haskell em 46º, ML em 48 º, enquanto Common Lisp está classificada entre as 50-100 mais utilizadas. Ademais, nos livros (MARLOW, 2013) e (PROKOPEC, 2014) são apresentados estudos mais aprofundados em relação a concorrência e programação paralela, principalmente sob linguagens regidas pela programação funcional.

A linguagem ALGOL teve bastante sucesso na época e apesar de não ser mais utilizada nos dias atuais, ela influenciou fortemente as linguagens imperativas subsequentes como Basic, Pascal e C, assim como, indiretamente, a maioria das linguagens imperativas atuais como C++, Java, C#, incluindo as linguagens de script como Perl, PHP e Javascript.

Outra linguagem que surgiu baseada em esforços de um comitê foi a linguagem COBOL, também no âmbito da programação imperativa. Diferente das outras linguagens imperativas que tinham um foco mais voltado para aplicações científicas, COBOL surgiu com um propósito comercial. Naquela época, era comum existir linguagens específicas para cada plataforma, o que motivou a criação de uma linguagem comum para aplicações comerciais. Curiosamente, COBOL foi uma das linguagens mais usadas na história da computação, apesar de ter tido pouca influência na criação de novas linguagens. Isso pode estar relacionado ao quão bem ela satisfez as necessidades dessa área de aplicação. Nos dias atuais, programas em COBOL ainda estão em uso globalmente em agências governamentais e militares além de empresas comerciais, incluindo bancos[28].

Durante o início da década de 70, outro paradigma entrou em cena, o paradigma lógico, com linguagens que o materializavam. Na verdade, inicialmente o projeto na época não tinha o foco de implementar uma linguagem de programação, mas realizar o processamento de linguagens naturais. Em suma, o objetivo era fazer uso de uma notação lógica formal para comunicar processos computacionais a um computador. Nesse âmbito, o Prolog foi a primeira linguagem a ser basear no paradigma lógico. Tal linguagem foi criada em meados de 1972, por Alain Colmerauer e Philippe Roussel, com base no conceito de interpretação procedimental das cláusulas de Horn[29], proposto por Robert Kowalski. O projeto foi motivado, em parte, pelo desejo de conciliar o uso da lógica como uma linguagem declarativa de representação do conhecimento. Nos dias de hoje, a linguagem Prolog tem sido

[28] De acordo com o TIOBE Index (2019), a linguagem COBOL está em 24º lugar na lista das linguagens mais utilizadas atualmente.

[29] A cláusula de Horn é uma fórmula lógica no formato de uma regra (condicional) que representa uma disjunção ou conjunção de literais. Elas são utilizadas na programação lógica para a demonstração de teoremas por meio da lógica de primeira ordem.

utilizada especialmente em projetos de IA como no IBM Watson para o processamento de linguagem natural[30].

Em 1974, mais da metade das aplicações de computadores no Departamento de Defesa dos EUA (*USA Department of Defence – DoD*, em inglês) era composta de sistemas embarcados (cujo hardware é embarcado no dispositivo que ele controla ou para o qual fornece serviços). Os custos de software elevaram-se rapidamente, isto por conta da crescente complexidade dos sistemas. Mais de 450 diferentes linguagens de programação estavam em uso em projetos do DoD e nenhuma delas era padronizada pelo DoD. Por causa dessa proliferação de linguagens, os softwares de aplicação raramente eram reutilizados.

Nesse âmbito, o Exército, a Marinha e a Força Aérea dos EUA propuseram o desenvolvimento de uma linguagem de alto nível para sistemas embarcados. A linguagem Ada é o resultado do mais extensivo e dispendioso esforço de projeto de uma linguagem já realizado na história da computação, tendo seu primeiro manual de referência publicado somente em 1979. Nos dias atuais, ela é utilizada em uma vasta gama de aplicações que dependem de sistemas críticos de segurança[31].

Neste meio tempo, na história da computação, outro paradigma vinha ganhando vida, o chamado paradigma orientado a objetos. Os conceitos formais da programação orientada a objetos foram introduzidos em meados de 1960 na linguagem SIMULA-67. Tal linguagem tinha a ideia geral de simular objetos do mundo real. Essa linguagem introduziu a noção de classes e de instâncias ou objetos como uma parte explícita da programação. A linguagem também implementou a coleta de

---

[30] Para competir com sucesso no Jeopardy! (jogo de perguntas e respostas criado por Merv Griffin, no qual os temas são apresentados como respostas e os concorrentes devem formular a pergunta correspondente a cada uma delas), o sistema Watson teve que responder a questões complexas de linguagem natural em um domínio de conhecimento extremamente amplo. Além disso, teve que calcular uma confiança precisa em suas respostas e completar seu processamento em um período muito curto de tempo. Para isso, o Watson utiliza a tecnologia de processamento de linguagem natural para interpretar a questão e extrair elementos-chave, como o tipo de resposta e as relações entre entidades. Essa parte do sistema foi desenvolvida com a linguagem Prolog, devido as características de simplicidade e expressividade inerentes ao paradigma lógico, permitindo converter facilmente as informações em fatos e regras da linguagem (LALLY e FODOR, 2011).

[31] Devido aos recursos de suporte a sistemas críticos de segurança fornecidos pela linguagem Ada, ela é usada não apenas para aplicações militares, mas também em projetos comerciais onde um bug de software pode ter graves consequências, por exemplo, na área da aviação e no controle de tráfego aéreo, em foguetes comerciais como Ariane 4 e 5, em satélites e outros sistemas espaciais, no transporte ferroviário e em sistemas bancários (FELDMAN, 2014).

lixo automática, conceito esse que havia sido introduzido em um momento anterior na linguagem LISP.

As ideias implementadas no SIMULA influenciaram fortemente a linguagem SmallTalk. A linguagem SmallTalk foi criada nos anos 70, na Xerox Palo Alto Research Center por um grupo de pesquisa liderado por Alan Kay. Ela é tida como uma das poucas linguagens orientadas a objetos realmente pura, na qual não há tipos primitivos e todos os elementos são construídos em forma de objetos (*i.e.*, números, classes, métodos, blocos de código etc.).

Outra linguagem de programação influenciada pela linguagem SIMULA, foi a linguagem C++. O C++ surgiu com as facilidades presentes no C para suportar grande parte daquilo em que o SIMULA e Smalltalk foram pioneiros. Em linhas gerais, o primeiro passo do C rumo ao C++ foi dado por Bjarne Stroustrup, do Bell Laboratories, em 1980. As modificações incluíram a adição da verificação de tipos dos parâmetros de função e conversão e, o que é mais significativo, o suporte a programação orientada a objetos. Desde aquela época a linguagem tem sido utilizada extensivamente[32] em aplicações diversas como em microcontroladores embarcados em eletrônicos, sistema de tempo real, sistemas de manufatura, jogos de videogame, dentre outros.

Ainda, no âmbito do paradigma orientado a objetos, a linguagem Java surgiu para vir a se tornar a linguagem de programação mais popular da história da computação[33]. Em geral, os projetistas do Java iniciaram seus estudos com base no C++, removeram numerosas construções, mudaram algumas e adicionaram outras. A linguagem resultante oferece grande parte do poder e da flexibilidade do C++, mas em uma linguagem mais simples e supostamente mais segura. Nesse sentido, a linguagem não fornece acesso a ponteiros de memória, mas em contrapartida oferece recursos como limpeza automática de lixo de memória e exceções. Essas funcionalidades são úteis para programação em mais alto nível, entretanto tiram a liberdade e, mesmo, acesso mais íntimo e direto a funções de mais baixo nível, o que pode ser um contraponto, justamente, em sistemas embarcados.

---

[32] Atualmente e por vários outros anos as linguagens C e C++ ficaram no top 2 e 3 de linguagens mais utilizadas, respectivamente (TIOBE, 2019).

[33] Nos últimos 15 anos, a linguagem Java foi e continua sendo a primeira no ranking das linguagens mais utilizadas no mundo, perdendo algumas poucas vezes a liderança para o C (TIOBE, 2019).

Na prática, a linguagem Java surgiu no início dos anos 90, quando a Sun Microsystems decidiu que nem C e nem C++ proviam o nível de confiabilidade necessário para embarcar programas em produtos eletrônicos de consumo, tais como torradeiras, fornos de micro-ondas e sistemas de TV interativos. Ainda que o impulso inicial do Java tenha sido para os produtos eletrônicos de consumo, nenhum dos produtos com os quais ele foi usado em seus primeiros anos chegou a ser comercializado. Foi durante a popularização da internet que o Java passou a ser considerado uma ferramenta útil para a programação da Web.

A primeira versão pública da linguagem foi introduzida em 1995, nomeada de Java 1.0. A linguagem promovia o slogan *Write Once, Run Everywhere* (Escreva uma vez e roda em qualquer lugar), provendo compiladores gratuitos para as principais plataformas de computadores da época. Nos dias atuais, Java é utilizada em larga escala em aplicações web e para o desenvolvimento de aplicativos para *smartphones* (plataforma Android).

A.2 Características para a avaliação de linguagens de programação

Assim como apresentado na subseção anterior, algumas das linguagens bem-sucedidas foram projetadas por indivíduos, outras por comitês de toda uma indústria e outras, ainda, foram produto de um forte apoio por parte de empresas, com âmbito comerciais. Assim, não fica claro se o processo do projeto, seja de forma individual, por meio de um comitê ou pela alavancagem corporativa, possui uma influência total sobre o sucesso de um projeto de linguagem (TUCKER e NOONAN, 2010).

Nesse âmbito, Tucker e Noonan (2010) elencaram algumas características que podem impactar no sucesso de uma boa linguagem de programação. Tais características são definidas como (a) simplicidade e legibilidade; (b) clareza nas ligações; (c) confiabilidade; (d) suporte; (e) abstração; (f) ortogonalidade; e (g) implementação eficiente. De modo a explorar tais características em maiores detalhes, as subseções seguintes apresentam a visão de ambos os autores em relação a definição de uma linguagem de programação de qualidade.

A.2.1 Simplicidade e legibilidade

De maneira geral, os programas devem apresentar facilidades de escrita, com um conjunto simplificado de instruções e, ao mesmo tempo, serem compreensíveis, no âmbito de facilitarem a leitura pelo programador mediano. Essas características são apontadas, tanto por (TUCKER e NOONAN, 2010) quanto por (SEBESTA, 2012), como um dos critérios mais importantes para julgar a qualidade de uma linguagem de programação.

Algumas linguagens, como Basic, Algol e Pascal, foram intencionalmente projetadas para facilitar a clareza de expressão. Basic, por exemplo, tinha um conjunto de instruções muito pequeno. Algol 60 tinha uma “linguagem de publicação” que fornecia um formato-padrão para a composição de programas que apareciam em artigos publicados. Pascal foi projetada explicitamente como uma linguagem para ensino, com recursos que facilitavam o uso de princípios da programação estruturada (TUCKER e NOONAN, 2010).

A partir de 1970, com os conceitos de ciclo de vida do software em alta, a manutenção de código foi reconhecida como uma parte principal do ciclo, particularmente em termos de custo. Como a facilidade de manutenção é determinada, em grande parte, pela legibilidade dos programas, tal medida se tornou importante na construção de novas linguagens (SEBESTA, 2012). Ademais, a simplicidade de uma linguagem de programação afeta fortemente sua legibilidade, influenciando diretamente na produtividade, depuração e manutenção de programas criados com tais linguagens.

Nesse sentido, linguagens com um conjunto de instruções grande tendem a ser mais difíceis de aprender. Outro problema ocorre quando uma linguagem apresenta mais de uma maneira de realizar uma mesma operação. Por exemplo, em C++ e Java, o programador pode incrementar uma variável inteira de quatro maneiras distintas, conforme apresenta o Código 70.

Código 70: Maneiras distintas de incrementar uma variável em C++ e em Java

```
1  count = count + 1;
2  count += 1;
3  count++;
4  ++count;
```

Fonte: Sebesta, 2012

Um terceiro problema é a sobrecarga de operadores, na qual um operador pode assumir mais de um significado. Apesar de sua utilidade, tal característica normalmente leva a uma redução de legibilidade na leitura de um programa. Em linguagens que o programador pode definir a sobrecarga de operadores, como em C++, a legibilidade fica comprometida e condicionada ao bom senso do desenvolvedor, o qual poderia ter dificuldades de aplicar tais operadores de maneira apropriada (SEBESTA, 2012).

A título de curiosidade, em Scriptol (2018) é possível conhecer brevemente a sintaxe de um conjunto de 200 diferentes linguagens de programação, comparando exemplos do clássico programa “Hello World” em cada qual, possibilitando aos desenvolvedores conhecer a expressividade das linguagens em um programa tradicionalmente usado como exemplo inicial. Assim como existem alguns exemplos de linguagens que expressam o programa em apenas uma única linha de código, tais como Abc, ActionScript, Apl, Lisp, Lua, Oz, Prolog e Python, existem linguagens que precisam de oito ou mais linhas para tal, como AspectJ, *Assembly* de diversas plataformas, Bliss, C++, Java e VHDL. Entretanto, é importante observar que o número de linhas deste programa básico não reflete necessariamente as facilidades de expressividade de cada linguagem, necessitando para tanto, um estudo mais aprofundado sobre as características das linguagens.

A.2.2 Clareza nas ligações no tocante ao sistema de tipos

De maneira geral, a tipagem de uma linguagem está condicionada à um conjunto de regras, chamada de sistema de tipos, que atribuem uma propriedade chamada de tipo para os construtos de um programa (*i.e*., variáveis, expressões, funções ou módulos). O objetivo principal do sistema de tipos é reduzir a possibilidade de erros em programas, por meio de uma interface entre variáveis e valores, garantindo que as partes foram conectadas de maneira apropriada e consistente (PIERCE, 2002).

A checagem de tipo pode acontecer estaticamente, em tempo de compilação, ou dinamicamente, durante a execução do programa. A verificação de tipos é um fator importante na confiabilidade da linguagem. Como a checagem de tipos em tempo de execução é considerada computacionalmente custosa, realizar esse procedimento em tempo de compilação, geralmente, é mais desejável. Além disso, quanto mais cedo

os erros nos programas forem detectados, menos dispendioso será para fazer os reparos necessários (SEBESTA, 2012).

Em qualquer caso, o sistema de tipos é parte da semântica da linguagem, sendo semântica de tempo de compilação no caso da tipagem estática e semântica de tempo de execução no caso da tipagem dinâmica. Na linguagem de programação Java, a checagem de tipo de quase todas as variáveis e expressões é feita em tempo de compilação, porém a conversão (*casting*) é verificada em tempo de execução (TUCKER e NOONAN, 2010).

Algumas linguagens como C e Ada, por exemplo, requerem que um único tipo seja associado a uma variável quando essa for declarada, permanecendo associada a esta durante toda sua vida em tempo de execução. Isso permite que o tipo de valor de uma expressão seja determinado em tempo de compilação (TUCKER e NOONAN, 2010).

De maneira geral, uma linguagem é considerada estaticamente tipada quando, em tempo de compilação, as variáveis de um determinado tipo são atribuídas única e exclusivamente a dados do mesmo tipo. Nesse sentido, o compilador garante que erros serão detectados antes do programa ser executado efetivamente (TUCKER e NOONAN, 2010).

De modo a apresentar a etapa de verificação de tipos, o Código 71 apresenta um exemplo de atribuição com tipos distintos.

Código 71: Exemplo de atribuição com tipos distintos

```
1  var x = 1;
2  x = "1";
```

Fonte: Autoria Própria

Conforme apresenta o Código 71, se for considerada uma linguagem tipada estaticamente, como a linguagem C# por exemplo, a atribuição da linha 2 resultaria em erro de compilação, pois o valor do tipo *string* está sendo atribuído a uma variável definida previamente (linha 1) como sendo do tipo *int*.

Outras linguagens como Perl, Python e Scheme, permitem que o tipo de uma variável seja redefinido cada vez que um novo valor for atribuído a ela em tempo de execução. Nesse âmbito, uma linguagem é considerada dinamicamente tipada quando o tipo de uma variável puder variar em tempo de execução de acordo com o valor atribuído (TUCKER e NOONAN, 2010).

Considerando o mesmo exemplo do Código 71, mas agora em uma linguagem dinamicamente tipada como o JavaScript (compatível sintaticamente com o exemplo em questão), a execução da linha 2 seria tratada como normal, não resultando em nenhum tipo de erro.

Independentemente de uma linguagem ser tipada estaticamente ou dinamicamente, a linguagem só é dita fortemente tipada se o seu sistema de tipo permitir que todos os erros em um programa sejam detectados em tempo de compilação ou em tempo de execução. Outrossim, a tipagem forte geralmente promove programas mais confiáveis e é vista como uma virtude no projeto de linguagens de programação (TUCKER e NOONAN, 2010).

Em Pierce (2002) são apresentados exemplos de linguagens estaticamente e fortemente tipadas: ML, Haskell, Java (quase totalmente) e Pascal (quase totalmente); linguagens estaticamente e fracamente tipadas: C e C++; linguagens dinamicamente e fortemente tipadas: Lisp e Scheme; e linguagens dinamicamente e fracamente tipadas: Perl.

De acordo com Tucker e Noonan (2010), uma boa linguagem deve ser muito clara sobre quando ocorre a ligação principal (*i.e*., quando um tipo é devidamente atribuído a um símbolo/variável) para cada elemento com sua propriedade. Nesse âmbito, são apresentados os principais tempos de ligação entre um elemento e sua propriedade, em uma ordem temporal (TUCKER e NOONAN, 2010):

- *Tempo de definição da linguagem*: quando a linguagem é definida, os tipos básicos de dados são ligados a suas palavras reservadas. Por exemplo, números inteiros são ligados ao identificador *int*, números reais são ligados ao identificador *float* e assim por diante;
- *Tempo de implementação da linguagem*: quando o compilador ou interpretador da linguagem é escrito, os valores são ligados a representações de máquina. Por exemplo, o tamanho de uma variável em bytes do tipo *int* é determinado no momento da implementação da linguagem;
- *Tempo de escrita do programa*: quando os programas são escritos em algumas linguagens que apresentam tipagem estática, os nomes de variáveis são ligados a tipos, os quais permanecem ligadas a esses durante toda a execução do programa;

- *Tempo de compilação*: quando os programas são compilados, os comandos e as expressões de tais programas são ligados a sequências de instruções em linguagem de máquina equivalente;
- *Tempo de carga do programa*: quando o código de máquina é carregado, as variáveis estáticas são atribuídas a endereços de memória, a pilha de tempo de execução é alocada a um bloco de memória, assim como o próprio código de máquina;
- *Tempo de execução do programa*: quando os programas são executados, as variáveis são ligadas a valores.

Nesse âmbito, um elemento pode ser ligado a uma determinada propriedade em qualquer um dos tempos citados. Tal ligação pode acontecer em tempo de compilação, durante a carga de um programa ou no início da execução do programa (TUCKER e NOONAN, 2010).

Ainda, segundo Tucker e Noonan (2010), dentro desse espectro de possibilidades, a noção de ligação precoce significa que um elemento é ligado a uma propriedade o mais rápido possível, em vez de mais tarde, nessa série de tempo. A ligação tardia significa um atraso até a última etapa possível. Nesse âmbito, a ligação precoce leva a uma melhor detecção de erros e geralmente é menos custosa. Todavia, ligações tardias levam a uma maior flexibilidade de programação.

De modo geral, um projeto de linguagem deve considerar todas essas alternativas e decisões sobre quando as ligações ocorrerem, o quão forte as tipagens devem ser, assim como a dinamicidade destas.

A.2.3 Confiabilidade

A tipagem forte impõem um certo impacto na confiabilidade de uma linguagem, uma vez que a execução de um programa está bem definida na compilação do mesmo. Tucker e Noonan (2010) defendem que um programa deveria se comportar da mesma forma quando executado sobre o mesmo conjunto de dados de entrada, mesmo em diferentes plataformas, sendo assim determinísticos nesses termos dados (TUCKER e NOONAN, 2010).

A capacidade de um programa interceptar erros em tempo de execução e outras condições incomuns detectadas pelo programa, bem como tomar medidas

corretivas e continuar a execução do mesmo são características que reforçam a confiabilidade de um programa. Como exemplo de linguagens que implementam um mecanismo de tratamento de exceções tem-se Ada, C++, Java e C#. Tais linguagens incluem grandes capacidades de manipular exceções. Em contraste, linguagens populares como C e Fortran não implementam tal funcionalidade (SEBESTA, 2012).

Outras funcionalidades desejáveis que restrinjam o uso de análises e vazamento de memória, que suportem tipagem forte, tenham sintaxe e semântica bem definidas e que suportem a verificação e validação de programas, afetam diretamente a confiabilidade de um programa, apresentando uma vantagem nessa característica (TUCKER e NOONAN, 2010).

A.2.4 Suporte

De acordo com Tucker e Noonan (2010), uma boa linguagem de programação deve ser facilmente acessível por alguém que queira aprendê-la e instalá-la em seu próprio computador. De forma ideal, seus compiladores devem ser de domínio público, em vez de serem propriedade da uma corporação, bem como devem ser implementados em múltiplas plataformas. Ademais, cursos, livros, tutoriais e um grande número de pessoas familiarizadas com a linguagem são vantagens que ajudam a preservar e estender a vitalidade de uma linguagem.

Questões relacionadas a custo podem ser de maior preocupação para alunos e programadores individuais, ao contrário de funcionários governamentais e corporativos, cujos custos de software são geralmente cobertos por suas empresas. A história das linguagens de programação tem apresentado sucesso em ambos os lados. Por exemplo, C, C++ e Java são linguagens não proprietárias, disponíveis no domínio público para uma ampla variedade de plataformas. Por outro lado, C# e Eiffel são linguagens suportadas por empresas cujo uso é restringido pelo seu custo e as plataformas/IDEs nas quais elas são implementadas (TUCKER e NOONAN, 2010).

Outro fator é o custo do sistema de implementação da linguagem. Um dos fatores que explicam a rápida aceitação da linguagem Java é que sistemas compiladores/interpretadores estavam à disposição para ela logo depois que seu projeto foi lançado pela primeira vez. Uma linguagem de programação cujo sistema de implementação seja caro ou somente execute sobre hardware caro, terá menos chances de se tornar popular (SEBESTA, 2012).

A.2.5 Abstração

Em suma, abstração, significa a capacidade de criar e definir módulos (*e.g.*, estruturas ou um conjunto de operações encapsuladas em funções/métodos), que podem ser utilizados ao longo de um programa, sem necessariamente se ater aos detalhes de seu funcionamento interno. A abstração é um aspecto fundamental no processo de projeto de linguagens de programação contemporâneas (SEBESTA, 2012).

Ademais, o grau de abstração permitido por uma linguagem de programação e a naturalidade de sua expressão, desempenham um papel importante em sua capacidade de escrita. As linguagens de programação podem suportar duas categorias distintas de abstração, a abstração de processo e a abstração de dados (SEBESTA, 2012).

Um exemplo simples de abstração de processo é o uso de um subprograma para implementar um algoritmo de classificação chamado diversas vezes em um programa. Sem a abstração de processo, o código de classificação teria de ser replicado em todos os lugares em que fosse necessário. De maneira geral, isso tornaria o programa muito mais longo e tendente a erros no caso de modificações no algoritmo. Além disso, tal programa se tornaria menos legível por ter os detalhes do algoritmo expostos ao longo do código (SEBESTA, 2012).

Como exemplo de abstração de dados, considere uma árvore binária que armazena dados inteiros em seus vértices. Em uma linguagem que não possui a característica de abstração de dados, tal implementação poderia ser feita com três vetores paralelos de números inteiros, em que dois dos inteiros seriam utilizados como subscritos para especificar os vértices descendentes. Em C++ e em Java, esses três vértices poderiam ser implementados usando-se de uma abstração de vértice da árvore em forma de uma classe simples com dois ponteiros e um número inteiro. A naturalidade da abordagem orientada a abstração de dados torna mais fácil de escrever e compreender um programa (SEBESTA, 2012).

De maneira geral, os programadores gastam muito tempo construindo código/algoritmos, tanto de dados quanto de processos. Nesse âmbito, o ideal é explorar a reutilização de código e evitar reinventá-lo. Ademais, bibliotecas que acompanham linguagens modernas de programação apresentam código pronto para construções comuns e úteis, desenvolvidas normalmente por programadores

experientes, o que evita a recodificação das mesmas. Por exemplo, as bibliotecas de classes Java contêm implementações de estruturas de dados básicas (como vetores e pilhas) que, em linguagens mais antigas, tinham de ser projetadas explicitamente pelos próprios desenvolvedores.

Em suma, uma boa linguagem de programação deveria suportar tão bem a abstração de dados, quanto a de processos (TUCKER e NOONAN, 2010).

A.2.6 Ortogonalidade

Uma linguagem é dita ortogonal quando um conjunto relativamente pequeno de construções primitivas (comandos e recursos) pode ser combinado a um número relativamente pequeno de formas para construir as estruturas de controle e de dados da linguagem. Além disso, cada possível combinação de primitivas é possível e significativa. Nesse âmbito, a ortogonalidade vem de uma simetria de relacionamentos entre primitivas (SEBESTA, 2012).

Como um exemplo de ortogonalidade, considere uma linguagem de programação que possibilite o uso de ponteiros. Tal linguagem deveria permitir que o ponteiro possa apontar para qualquer tipo específico definido na linguagem, assim como é possível em C/C++, por exemplo. Entretanto, se não for permitido aos ponteiros apontar para vetores ou para objetos de classes definidas pelos programadores, muitas construções potencialmente úteis não poderiam ser definidas, limitando o uso da linguagem (SEBESTA, 2012).

Em contraste, as linguagens C/C++ também apresentam exemplos de falta de ortogonalidade, como por exemplo a possibilidade de registros (*struct*) poderem ser retorno de uma chamada de função, mas elementos do tipo vetor (*arrays*) não. Nesse mesmo âmbito, parâmetros são passados por valor, a menos que sejam *arrays* que obrigatoriamente são passados por referência (SEBESTA, 2012). Outras linguagens restringem os tipos de objetos que podem ser passados em uma chamada. Por exemplo, a maioria das linguagens imperativas não permite que definições de funções sejam passadas como argumentos e, portanto, não são ortogonais em relação a isso (TUCKER e NOONAN, 2010).

A ortogonalidade está intimamente relacionada à simplicidade. Nesse sentido, quanto mais ortogonal for uma linguagem, menos regras excepcionais são necessárias para a escrita de programas corretos. Além disso, menos exceções

significam um maior grau de regularidade no projeto. Assim, programas em uma linguagem ortogonal, muitas vezes, tendem a ser mais simples e mais claros do que aqueles construídos em uma linguagem não-ortogonal (TUCKER e NOONAN, 2010; SEBESTA, 2012).

De maneira geral, a ortogonalidade tende a se correlacionar com a simplicidade conceitual, já que o programador não precisa lidar com muitas regras excepcionais durante a escrita de programas. Por outro lado, a não-ortogonalidade, muitas vezes está relacionada à eficiência porque suas regras excepcionais eliminam as opções de programação que consumiriam tempo ou espaço (TUCKER e NOONAN, 2010).

A.2.7 Implementação eficiente

Os recursos e as construções de uma linguagem devem permitir uma implementação prática e eficiente em plataformas contemporâneas. Conforme apresentado em Tucker e Noonan (2010) a linguagem de programação Algol 68 era um projeto de linguagem elegante, mas suas especificações eram tão complexas que era (quase) impossível implementá-las efetivamente.

Como outro exemplo, versões iniciais de Ada foram criticadas por suas características de tempo de execução ineficientes, uma vez que Ada foi projetada, em parte, para suportar programas que seriam executados em "tempo real". De fato, o custo para executar programas é bastante influenciado pelo projeto da linguagem. Se tal linguagem exigir demasiadas verificações de tipos durante a execução, comprometerá o tempo de execução de programas, independente da qualidade do compilador (SEBESTA, 2012).

APÊNDICE B

BNF DA LINGUAGEM DE PROGRAMAÇÃO LINGPON PRELIMINAR

Código 72: BNF da linguagem de programação LingPON Preliminar

```
1  <program>               ::= <fbes> <inst> <rules>
2
3  <fbes>                  ::= <fbe>
4                           |  <fbe> <fbes>
5
6  <fbe>                   ::= FBE <id> <fbe_body> END_FBE
7                           |  FBE <fbe_body> END_FBE
8
9  <id>                    ::= ID
10                          |  ID POINT ID
11
12 <fbe_body>              ::= <decl_attributes> <decl_methods>
13
14 <decl_attributes>       ::= ATTRIBUTES <atributes> END_ATTRIBUTES
15
16 <atributes>             ::= <attributes_body>
17                          |  <attributes_body> <attributes>
18
19 <attributes_body>       ::= <type> <id> <value>
20
21 <type>                  ::= BOOLEAN
22                          |  INTEGER
23                          |  REAL
24                          |  STRING
25                          |  <id>
26
27 <value>                 ::= NUMBER
28                          |  <boolean>
29                          |  <id>
30
31 <boolean>               ::= TRUE
32                          |  FALSE
33
34 <decl_methods>          ::= METHODS <methods> END_METHODS
35
36 <methods>               ::= <method_body>
37                          |  <method_body> <methods>
38
39 <method_body>           ::= METHOD <id> LP <id> ASSIGN <value> RP
40
41 <inst>                  ::= INST <declarations> END_INST
42
43 <declarations>          ::= <declaration>
44                          |  <declaration> <declarations>
45
46 <declaration>           ::= <type> <ids>
47
48 <ids>                   ::= <id>
49                          |  <id> COMMA <ids>
50
51 <rules>                 ::= <rule>
52                          |  <rule> <rules>
53
54 <rule>                  ::= RULE <rule_body> END_RULE
55                          |  RULE <id> <rule_body> END_RULE
56
57 <rule_body>             ::= <decl_condition> <decl_action>
58
59 <decl_condition>        ::= CONDITION <condition_body> END_CONDITION
```

```
                          |  CONDITION <id> <condition_body> END_CONDITION

<condition_body>         ::= <subcondition> <operator> <condition_body>
                          |  <subcondition>

<operator>               ::= AND
                          |  OR

<subcondition>           ::= SUBCONDITION <subcondition_body>
                                   END_SUBCONDITION
                          |  SUBCONDITION <id> <subcondition_body>
                                   END_SUBCONDITION

<subcondition_body>      ::= <premise> AND <subcondition_body>
                          |  <premise>

<premise>                ::= PREMISE <exp>
                          |  PREMISE <id> <exp>

<exp>                    ::= <fator> <comp> <fator>

<fator>                  ::= <id>
                          |  NUMBER
                          |  boolean

<comp>                   ::= EQ
                          |  NE
                          |  LT
                          |  GT
                          |  LE
                          |  GE

<decl_action>            ::= ACTION <action_body> END_ACTION
                          |  ACTION <id> <action_body> END_ACTION

<action_body>            ::= <action_elements> <action_body>
                          |  <action_elements>

<action_elements>        ::= <instigation>
                          |  <method_use>
                          |  <exp> SEMICOLON

<instigation>            ::= INSTIGATION <method_use>
                          |  INSTIGATION <id> <method_use>

<method_use>             ::= <id> LP RP SEMICOLON
```

Fonte: Adaptado de Ronszcka *et al.*, 2013

APÊNDICE C

BNF DA LINGUAGEM DE PROGRAMAÇÃO LINGPON 1.0

Código 73: BNF da linguagem de programação LingPON 1.0

```
<program>               ::= <fbes> <inst> <strategy> <rules> <main>
                          | <fbes> <inst> <strategy> <rules>

<fbes>                  ::= <fbe>
                          | <fbe> <fbes>

<fbe>                   ::= FBE <id> <fbe_body> END_FBE

<inst>                  ::= INST <declarations> END_INST

<strategy>              ::= STRATEGY <strategy_declaration> END_STRATEGY

<strategy_declaration>  ::= NO_ONE
                          | BREADTH
                          | DEPTH

<declarations>          ::= <declaration>
                          | <declaration> <declarations>

<declaration>           ::= <type> <ids>

<ids>                   ::= <id>
                          | <id> COMMA <ids>

<rules>                 ::= <rule>
                          | <rule> <rules>

<rule>                  ::= RULE <rule_body> END_RULE
                          | RULE <id> <rule_body> END_RULE

<rule_body>             ::= <decl_condition> <decl_action>
                          | <decl_properties> <decl_condition> <decl_action>

<decl_properties>       ::= PROPERTIES <properties_body> END_PROPERTIES

<properties_body>       ::= <properties_type> <value>

<properties_type>       ::= PRIORITY
                          | KEEPER

<decl_condition>        ::= CONDITION <condition_body> END_CONDITION
                          | CONDITION <id> <condition_body> END_CONDITION

<condition_body>        ::= <subcondition> <operator> <condition_body>
                          | <subcondition>

<operator>              ::= AND
                          | OR

<subcondition>          ::= SUBCONDITION <id> <subcondition_body> END_SUBCONDITION

<subcondition_body>     ::= <premise> AND <subcondition_body>
                          | <premise>

<premise>               ::= PREMISE <exp>
                          | PREMISE <id> <exp>
                          | PREMISE IMP <exp>
                          | PREMISE IMP <id> <exp>

```

```
<exp>                   ::= <fator> <comp> <fator>

<comp>                  ::= EQ
                         |  NE
                         |  LT
                         |  GT
                         |  LE
                         |  GE

<fator>                 ::= <id>
                         |  NUMBER
                         |  boolean
                         |  FLOATVALUE
                         |  STRINGVALUE
                         |  CHARVALUE

<boolean>               ::= TRUE
                         |  FALSE

<decl_action>           ::= ACTION <action_body> END_ACTION
                         |  ACTION <id> <action_body> END_ACTION

<action_body>           ::= <action_elements> <action_body>
                         |  <action_elements>

<action_elements>       ::= <instigation>
                         |  <method_use>
                         |  <exp> SEMICOLON

<instigation>           ::= INSTIGATION <method_use>
                         |  INSTIGATION <id> <method_use>

<method_use>            ::= <id> LP RP SEMICOLON

<id>                    ::= ID
                         |  ID POINT <id>

<decl_attributes>       ::= ATTRIBUTES <attributes> END_ATTRIBUTES

<attributes>            ::= <attributes_body>
                         |  <attributes_body> <attributes>

<attributes_body>       ::= <type> <id> <value>
                         |  <type> <id> SEMICOLON

<type>                  ::= BOOLEAN
                         |  INTEGER
                         |  PFLOAT
                         |  STRING
                         |  CHAR

<value>                 ::= NUMBER
                         |  FLOATVALUE
                         |  STRINGVALUE
                         |  CHARVALUE
                         |  <boolean>

<decl_methods>          ::= METHODS <methods> END_METHODS

<methods>               ::= <method_body>
                         |  <method_body> <methods>

<method_body>           ::= METHOD <id> LP <id> ASSIGN <id>
                                <method_operator> <value> RP
                         |  METHOD <id> LP <id> ASSIGN <id>
                                <method_operator> <id> RP
                         |  METHOD <id> LP <id> ASSIGN <value RP
                         |  METHOD <id> LP <id> ASSIGN <id> RP
```

```
128                             |  METHOD <id> LP <id> RP INNER_CODE_METHOD
129                             |  METHOD <id> LP RP INNER_CODE_METHOD
130
131  <method_operator>         ::= PLUS
132                             |  MINUS
133                             |  MULT
134                             |  DIV
135
136  <main>                    ::= MAIN INNER_CODE_MAIN
```

Fonte: Adaptado de Ferreira, 2015

APÊNDICE D

BNF DA LINGUAGEM DE PROGRAMAÇÃO LINGPON 1.2

Código 74: BNF da linguagem de programação LingPON 1.2

```
1  <program>               ::= <fbes> <inst> <strategy> <rules> <main>
2                           |  <fbes> <inst> <strategy> <rules>
3
4  <fbes>                  ::= <fbe>
5                           |  <fbe> <fbes>
6
7  <fbe>                   ::= FBE <id> <fbe_body> END_FBE
8                           |  FBE <id> <extends> <id> <fbe_body> END_FBE
9
10 <fbe_body>              ::= <decl_attributes> <decl_methods> <fbeRules>
11                          |  <decl_attributes> <decl_methods>
12                          |  <decl_attributes>
13
14 <inst>                  ::= INST <declarations> END_INST
15
16 <strategy>              ::= STRATEGY <strategy_declaration> END_STRATEGY
17
18 <strategy_declaration>  ::= NO_ONE
19                          |  BREADTH
20                          |  DEPTH
21
22 <declarations>          ::= <declaration>
23                          |  <declaration> <declarations>
24
25 <declaration>           ::= <type> <ids>
26
27 <ids>                   ::= <id>
28                          |  <id> COMMA <ids>
29
30 <rules>                 ::= <rule>
31                          |  <rule> <rules>
32                          |  <formRule>
33                          |  <formRule> <rules>
34
35 <rule>                  ::= RULE <rule_body> END_RULE
36                          |  RULE <id> <rule_body> END_RULE
37                          |  RULE <depends> <id> <rule_body> END_RULE
38                          |  RULE <id> <depends> <id> <rule_body> END_RULE
39
40 <formRule>              ::= FORM_RULE <rule_body> END_FORM_RULE
41                          |  FORM_RULE <id> <rule_body> END_FORM_RULE
42                          |  FORM_RULE <depends> <id> <rule_body> END_FORM_RULE
43                          |  FORM_RULE <id> <depends> <id> <rule_body> END_FORM_RULE
44
45 <fbeRules>               ::= <fbeRule>
46                          |  <fbeRule> <fbeRules>
47
48 <fbeRule>                ::= FBE_RULE <rule_body> END_FBE_RULE
49                          |  FBE_RULE <id> <rule_body> END_FBE_RULE
50                          |  FBE_RULE <depends> <id> <rule_body> END_FBE_RULE
51                          |  FBE_RULE <id> <depends> <id> <rule_body> END_FBE_RULE
52
53 <depends>                ::= DEPENDS
54
55 <extends>                ::= EXTENDS
56
57 <rule_body>              ::= <decl_condition> <decl_action>
58                          |  <decl_properties> <decl_condition> <decl_action>
59
```

```
<decl_properties>       ::= PROPERTIES <properties_body> END_PROPERTIES

<properties_body>       ::= <properties_type> <value>

<properties_type>       ::= PRIORITY
                         |  KEEPER

<decl_condition>        ::= CONDITION <condition_body> END_CONDITION
                         |  CONDITION <id> <condition_body> END_CONDITION

<condition_body>        ::= <subcondition> <operator> <condition_body>
                         |  <subcondition>

<operator>              ::= AND
                         |  OR

<subcondition>          ::= SUBCONDITION <id> <subcondition_body> END_SUBCONDITION

<subcondition_body>     ::= <premise> AND <subcondition_body>
                         |  <premise>

<premise>               ::= PREMISE <exp>
                         |  PREMISE <id> <exp>
                         |  PREMISE IMP <exp>
                         |  PREMISE IMP <id> <exp>

<exp>                   ::= <fator> <comp> <fator>

<comp>                  ::= EQ
                         |  NE
                         |  LT
                         |  GT
                         |  LE
                         |  GE

<fator>                 ::= <id>
                         |  NUMBER
                         |  boolean
                         |  FLOATVALUE
                         |  STRINGVALUE
                         |  CHARVALUE

<boolean>               ::= TRUE
                         |  FALSE

<decl_action>           ::= ACTION <action_body> END_ACTION
                         |  ACTION <id> <action_body> END_ACTION

<action_body>           ::= <action_elements> <action_body>
                         |  <action_elements>

<action_elements>       ::= <instigation>
                         |  <method_use>
                         |  <exp> SEMICOLON

<instigation>           ::= INSTIGATION <method_use>
                         |  INSTIGATION <id> <method_use>

<method_use>            ::= <id> LP RP SEMICOLON

<id>                    ::= ID
                         |  ID POINT <id>

<decl_attributes>       ::= ATTRIBUTES <attributes> END_ATTRIBUTES

<attributes>            ::= <attributes_body>
                         |  <attributes_body> <attributes>
```

```
128 <attributes_body>       ::= <type> <id> <value>
129                          |  <type> <id> SEMICOLON
130
131 <type>                   ::= BOOLEAN
132                          |  INTEGER
133                          |  PFLOAT
134                          |  STRING
135                          |  CHAR
136                          |  <id>
137
138 <value>                  ::= NUMBER
139                          |  <boolean>
140                          |  <id>
141                          |  FLOATVALUE
142                          |  STRINGVALUE
143                          |  CHARVALUE
144
145 <decl_methods>           ::= METHODS <methods> END_METHODS
146
147 <methods>                ::= <method_body>
148                          |  <method_body> <methods>
149
150 <method_body>            ::= METHOD <id> LP <id> ASSIGN <id>
151                                  <method_operator> <value> RP
152                          |  METHOD <id> LP <id> ASSIGN <id>
153                                  <method_operator> <id> RP
154                          |  METHOD <id> LP <id> ASSIGN <value RP
155                          |  METHOD <id> LP <id> ASSIGN <id> RP
156                          |  METHOD <id> LP <id> RP INNER_CODE_METHOD
157                          |  METHOD <id> LP RP INNER_CODE_METHOD
158
159 <method_operator>        ::= PLUS
160                          |  MINUS
161                          |  MULT
162                          |  DIV
163
164 <main>                   ::= MAIN INNER_CODE_MAIN
```

Fonte: Adaptado de Santos, 2017

APÊNDICE E

PRINCIPAIS DISCUSSÕES NO GRUPO FOCAL

Em suma, no dia 5 de junho de 2018 às 19:00, 10 pesquisadores especialistas ou conhecedores do PON, além do autor desta tese, aqui citado como Ronszcka, participaram do grupo focal com o objetivo de avaliar o método proposto nesta tese, bem como o conjunto de tecnologias associadas. Tais pesquisadores são a seguir listados e citados posteriormente por seus sobrenomes:

- Jean Marcelo **Simão** – Professor da UTFPR – Campus Curitiba. Foi o criador do PON e ajudou na proposição da linguagem e do compilador para o PON, além de orientar ou coorientar todos os trabalhos relevantes do PON;
- João Alberto **Fabro** – Professor da UTFPR – Campus Curitiba. Ajudou na proposição da linguagem e do compilador para o PON, além de orientar e coorientar trabalhos relacionados em LingPON e PON em geral;
- Ricardo **Kerschbaumer** – Professor do Instituto Federal Catarinense (IFC) – então doutorando em PON com trabalho em Tecnologia PON-HD. Ele trabalhou na geração de código para VHDL, mais precisamente para o PON-HD;
- Fernando **Schütz** – Professor da UTFPR – Campus Medianeira. Doutorando em PON com o trabalho relacionado a NeuroPON. Trabalhou na versão *Static* do compilador e contribuiu com modificações na linguagem;
- Igor Thiago Marques **Mendonça** – então doutorando em PON com trabalho em Modelagem Orientada a Notificações (MON);
- Leonardo Faix **Pordeus** – então mestre em PON, atualmente doutorando em PON. No mestrado ele trabalhou com a geração de código para NOCA e contribuiu com modificações na linguagem;
- Robson **Xavier** – Mestre em PON – Trabalhou com a comparação entre POE e PON, bem como a classificação do PON via classificação de paradigmas de Peter van Roy;

- Marcos **Talau** - Professor da UTFPR – Campus Dois Vizinhos – doutorando na área de redes de computadores. Ele trabalhou com o Framework PON C++ para ambientes distribuídos, via sockets UDP/TCP, chamado PON-IP;
- Luiz Fernando **Copetti** – Professor da UTFPR – Campus Curitiba – na época já interessado em doutorado no PON – com experiência em VHDL e NOCA;
- Christian Carlos Souza **Mendes** – Professor da UTFPR – Campus Curitiba – na época já interessado em doutorado no PON – tinha algum conhecimento em PON.

Em suma, a discussão se iniciou com Ronszcka incitando o grupo sobre a possibilidade do Grafo PON ser o elo de ligação entre todos os trabalhos que já foram criados, até então, pelo grupo de pesquisa, bem como dos que viriam a ser criados futuramente. Além disso, foi incitado o quanto a linguagem do PON (LingPON) poderia estar relacionada ou, até mesmo, influenciar trabalhos paralelos outros, como os ligados a Inteligência Artificial (IA) e Inteligência Computacional (IC), por exemplo. Mendonça concordou e discorreu de algumas questões relacionadas ao MON (Modelagem Orientada a Notificações) e sua possível relação com o Grafo PON.

Simão indagou que o trabalho em si não estaria apenas relacionado a linguagem, porém ao próprio PON, e o quanto o Grafo PON implicaria na corroboração do PON em termos de suas propriedades elementares. A linguagem seria apenas uma forma de expressar linguisticamente o Grafo PON em alto nível (1ª propriedade elementar). Ainda, Simão acrescentou que o PON é baseado em grafos, diferentemente dos demais paradigmas que se baseiam em árvore para expressar seus relacionamentos. Nesse âmbito, a linguagem estaria relacionada a programação em alto nível, enquanto o grafo estaria relacionado ao desacoplamento (2ª propriedade elementar). Por fim, a especialização do grafo para plataformas distintas permitiria explorar o desempenho inerente do modelo não-redundante (3ª propriedade elementar).

A discussão seguiu em torno do Grafo PON preliminar e Fabro salientou que o Grafo PON deveria ser mais do que uma simples estrutura de dados encapsulada em objeto e seus agregados, mas sim uma ferramenta para auxiliar e simplificar a etapa de navegação e extração dos dados do mesmo. Com isso, tal ferramenta deveria contar com funções que retornariam o número de entidades de determinado

tipo, quais entidades de determinado tipo fazem conexão com entidades de outro tipo etc. Na prática, um dos maiores problemas da versão preliminar do Grafo PON foi justamente essa falta de um “começo, meio e fim” na estrutura, que impediu os desenvolvedores de explorarem o grafo tão adequadamente quanto poderiam no âmbito de geração de código para os *targets* para os quais trabalharam. A maior dificuldade estava na incerteza dos desenvolvedores sobre a correta utilização do grafo em seus geradores de código, o que não raro levava a busca de tutoriamento ao seu autor. Tal constatação foi confirmada por Kerschbaumer, que explicitou que precisou fazer repetidas iterações na estrutura para localizar as informações necessárias.

Ronszcka acrescentou que o Grafo PON também poderia auxiliar na validação da linguagem, seja por meio de testes de software ou pequenos programas para validar sua plenitude em relação aos *targets* desenvolvidos. Fabro complementou que uma das dificuldades da programação em PON em si é a falta de depuradores de código e que o próprio ferramental em torno do Grafo PON poderia se tornar um interpretador, que executaria regras online e iterativamente mostraria o fluxo de execução do programa.

Schütz e Pordeus destacaram que sentiram falta de conceitos não implementados na linguagem que deveriam ser implementados em seus *targets*, como o gerador de código para a versão C++ *Static* no caso do Schütz e o gerador de código para versão *AssemblyPON* da NOCA no caso de Pordeus. Pordeus ainda destacou que o NOCA possuía conceitos implementados e disponíveis em seu AssemblyPON que não puderam ser implementados pela falta de plenitude da linguagem e do Grafo PON preliminares.

Ronszcka continuou a discussão em termos de modularização do código escrito em LingPON, contando que este foi um dos temas levantados pela banca de qualificação de doutorado. Ronszcka apresentou ao grupo uma proposta de separação das entidades globais da linguagem, que na época se apresentavam esparsas no código, propondo grupos para a organização de *FBEs* e *Rules*. Apesar disso, um programa PON ainda era um “arquivo unívoco”, contendo toda a especificação de um programa. A discussão continuou em torno da possibilidade de separar um programa PON em arquivos. Simão sugeriu a ideia de nomear os arquivos e tratá-los como os *Namespaces* em C++. Copetti e Kerschbaumer deliberaram sobre a possibilidade de definição de escopos, assim como em VHDL, permitindo criar

módulos e utilizá-los ao longo da definição de programas escritos em tal linguagem. Algumas discussões técnicas acerca dos termos a serem utilizados na definição dos escopos e encapsulamentos teve dois vieses, Copetti e Kerschbaumer defenderam algumas questões de modularização em hardware enquanto Mendes e Talau defenderam que desenvolvedores em geral estão acostumados com a modularização em termos de “público” e “privado”, comum em abordagens de software.

A conclusão da discussão em torno de modularização foi essencialmente a definição de um modelo de programação holônica (ou todo-parte). Simão comentou que se deveria aproveitar as “partes pertinentes” da programação orientada a objetos, como o encapsulamento, buscando a coesão e desacoplamento modular para com os *FBEs*. Simão continuou sobre a possibilidade de trabalhar com estados internos e externos a um *FBE*, de forma que *Rules* internas a um dado *FBE* poderiam executar em escopo local com base em *Attributes* internos (ou privados) e externar (ou tornar públicos) apenas o estado de *Attributes* pertinentes ao “mundo externo”. Na prática, o programa PON seria composto por um *FBE* principal e por instâncias de *FBEs* internos, que também poderiam ter *N* níveis de composição de outros *FBEs*.

Kerschbaumer acrescentou que a linguagem VHDL funciona de forma parecida, com a ideia de entradas e saídas (*in* e *out*). Copetti complementou que em VHDL a porta de um programa é global e o sinal é local, apresentando conceitos similares aos vislumbrados para o PON. Ronszcka concluiu, portanto, que a modularização simplificaria de fato o problema da LingPON em relação a *FBEs* e *Rules* globais, garantindo de certa forma mais coesão e desacoplamento dos entes de um programa. Simão complementou que assim os programas em PON seriam “autocontidos”, onde pequenas “esferas de notificação” teriam seus processamentos internos e apenas notificariam externamente quando necessário, corroborando com o alcance de coesão e desacoplamento de fato. Fabro, por fim, acrescentou que tal abordagem facilitaria inclusive a distribuição, onde estes pontos de ligação no grafo poderiam definir os pontos de distribuição em nós computacionais e as conexões entre estes.

Em outro momento, Ronszcka apresentou um novo tema, apontando uma sugestão para a criação de propriedades internas aos *FBEs*, o que, inclusive, corrobora com a discussão anterior. Nessa nova seção seria possível definir os mecanismos de escalonamento e resolução de conflitos, bem como o *target* de compilação para cada *FBE* individualmente. Simão comenta que este último assunto

seria justamente um trabalho futuro em andamento, proposta de tese de Pordeus, o qual faria esta integração entre plataformas distintas, por meio de um mesmo programa PON compilado. Simão acrescentou que no projeto de doutoramento de Pordeus, a proposta seria expressar por meio da engenharia de sistemas, os requisitos em regras, sendo que os requisitos não-funcionais, como tempo de resposta, por exemplo, poderia direcionar a compilação de trechos de um programa para *FPGA* e outras partes, que dependeriam de mais dinamicidade, por exemplo, para uma compilação em software.

Outro tema importante apontado por Ronszcka na discussão foi o de compatibilidade e integração com códigos legados e implementados sob os princípios de outros paradigmas. Para isso, Ronszcka apresentou uma proposta de inclusão de blocos e métodos para C++, que poderiam “enxertar” trechos ou chamadas de métodos de objetos de C++ na própria escrita de um programa PON, de modo a permitir que a compilação integre o código PON gerado com o código “enxertado” em um mesmo programa alvo. Fabro sugeriu que que essa mesma ideia poderia ser utilizada para qualquer linguagem de programação alvo, possibilitando que o desenvolvedor possa informar qualquer linguagem para o bloco “enxertado”. Fabro complementou que um mesmo programa poderia ter vários blocos, um para cada linguagem e *target*, que seria escolhido apropriadamente pelo compilador, de acordo com o *target* escolhido, possibilitando assim uma compilação para plataformas distintas de um mesmo programa especializado. Xavier acrescentou que em algumas abordagens orientas a evento, o próprio compilador faz o “enxerto” de blocos de código de modo a organizar a execução dos mesmos em linguagens de baixo nível.

APÊNDICE F

BNF DA LINGUAGEM DE PROGRAMAÇÃO NOPL

Código 75: BNF da linguagem de programação NOPL

```
 1 <program>               ::= FBE <id> <fbe_body> END_FBE
 2
 3 <fbe_body>              ::= <include_block>
 4                           |  <include_block> <fbe_body>
 5                           |  <properties_block>
 6                           |  <properties_block> <fbe_body>
 7                           |  <instance>
 8                           |  <instance> <fbe_body>
 9                           |  <attribute>
10                           |  <attribute> <fbe_body>
11                           |  <vector_instance>
12                           |  <vector_instance> <fbe_body>
13                           |  <vector_attribute>
14                           |  <vector_attribute> >fbe_body>
15                           |  <method>
16                           |  <method> <fbe_body>
17                           |  <rule>
18                           |  <rule> <fbe_body>
19                           |  <formation_rule>
20                           |  <formation_rule> <fbe_body>
21                           |  <main_block>
22                           |  <main_block> <fbe_body>
23
24 <include_block>          ::= INCLUDES <target> END_INCLUDES
25
26 <properties_block>       ::= PROPERTIES <properties> END_PROPERTIES
27
28 <properties>             ::= <property>
29                           |  <property> <properties>
30
31 <property>               ::= TARGET <target>
32                           |  STRATEGY <strategy>
33
34 <target>                 ::= CODE_GENERATION_EXAMPLE
35                           |  NAMESPACES
36                           |  FRAMEWORK_CPP_2_0
37                           |  FRAMEWORK_CPP_3_0
38
39 <strategy>               ::= NO_ONE
40                           |  BREADTH
41                           |  DEPTH
42                           |  PRIORITY
43
44 <instance>               ::= <visibility> <id> <id>
45
46 <attribute>              ::= <visibility> <type> <id> ASSIGN <factor>
47
48 <vector_instance>        ::= <visibility> <id> LB INTEGER_VALUE RB <id>
49
50 <vector_attribute>       ::= <visibility> <type> LB INTEGER_VALUE RB <id>
51
52 <vector_attribute>       ::= <visibility> <type> LB INTEGER_VALUE RB <id>
53                                ASSIGN LC <array_factor> RC
54
55 <method>                 ::= <visibility> METHOD <id> <method_body> END_METHOD
56
57 <method_body>            ::= <code_blocks>
58                           |  <params_body> <attribution>
59                           |  <params_body> <code_blocks>
```

```
60                          |  <attribution>
61
62  <params_body>           ::= PARAMS <params> END_PARAMS
63
64  <params>                ::= <param>
65                          |  <param> <params>
66
67  <param>                 ::= <type> <id>
68
69  <code_blocks>           ::= <code_block>
70                          |  <code_block> <code_blocks>
71
72  <code_block>            ::= CODE <target> END_CODE
73
74  <attribution>           ::= ATTRIBUTION <element> ASSIGN <factor> END_ATTRIBUTION
75
76  <rule>                  ::= RULE <id> <rule_properties_block> <condition>
77                                 <action> END_RULE
78                          |  RULE <id> <condition> <action> END_RULE
79
80  <formation_rule>        ::= FORMATION_RULE <id> <rule_indexes>
81                                 <rule_properties_block> <condition>
82                                 <action> END_FORMATION_RULE
83                          |  FORMATION_RULE <id> <rule_indexes> <condition>
84                                 <action> END_FORMATION_RULE
85
86  <rule_indexes>          ::= <rule_index>
87                          |  <rule_index> <rule_indexes>
88
89  <rule_index>            ::= INDEX <id> FROM INTEGER_VALUE TO INTEGER_VALUE
90
91  <rule_properties_block> ::= PROPERTIES <rule_properties> END_PROPERTIES
92
93  <rule_properties>       ::= <rule_property>
94                          |  <rule_property> <rule_properties>
95
96  <rule_property>         ::= PRIORITY INTEGER_VALUE
97                          |  KEEPER <boolean>
98
99  <condition>             ::= CONDITION <subconditions> END_CONDITION
100                         |  CONDITION <id> <subconditions> END_CONDITION
101                         |  CONDITION <premises> END_CONDITION
102                         |  CONDITION <id> <premises> END_CONDITION
103
104 <subconditions>         ::= <subcondition>
105                         |  <subcondition> <conjunction> <subconditions>
106
107 <subcondition>          ::= SUBCONDITION <premises> END_SUBCONDITION
108                         |  SUBCONDITION <id> <premises> END_SUBCONDITION
109
110 <premises>              ::= <premise>
111                         |  <premise> <conjunction> <premises>
112
113 <premise>               ::= PREMISE <expression> END_PREMISE
114                         |  PREMISE <id> <expression> END_PREMISE
115                         |  PREMISE IMPERTINENT <expression> END_PREMISE
116                         |  PREMISE IMPERTINENT <id> <expression> END_PREMISE
117
118 <expression>            ::= <factor> <symbol> <factor>
119
120 <symbol>                ::= EQ
121                         |  NE
122                         |  LT
123                         |  GT
124                         |  LE
125                         |  GE
126
127 <action>                ::= ACTION <execution> <instigations> END_ACTION
```

```
128                          |  ACTION <id> <execution> <instigations> END_ACTION
129                          |  ACTION <instigations> END_ACTION
130                          |  ACTION <id> <instigations> END_ACTION
131
132 <instigations>          ::= <instigation>
133                          |  <instigation> <instigations>
134
135 <instigation>           ::= INSTIGATION <execution> <calls> END_INSTIGATION
136                          |  INSTIGATION <id> <execution> <calls> END_INSTIGATION
137                          |  INSTIGATION <calls> END_INSTIGATION
138                          |  INSTIGATION <id> <calls> END_INSTIGATION
139
140 <execution>             ::= SEQUENTIAL
141                          |  PARALLEL
142
143 <calls>                 ::= <call>
144                          |  <call> <calls>
145
146 <call>                  ::= CALL <elementcall>
147
148 <elementcall>           ::= <id> POINT <id> LP RP
149                          |  THIS POINT <id> LP RP
150                          |  <id> POINT <id> LP <arguments> RP
151                          |  THIS POINT <id> LP <arguments> RP
152
153 <arguments>             ::= <argument>
154                          |  <argument> COMMA <arguments>
155
156 <argument>              ::= <factor>
157
158 <type>                  ::= <basictype>
159                          |  <supertype>
160
161 <supertype>             ::= <basictype> LT INTEGER_VALUE GT
162
163 <basictype>             ::= BOOLEAN
164                          |  INTEGER
165                          |  DOUBLE
166                          |  STRING
167                          |  CHAR
168
169 <visibility>            ::= PRIVATE
170                          |  PUBLIC
171
172 <conjunction>           ::= AND
173                          |  OR
174
175 <array_factor>          ::= <array_factor> COMMA <factor>
176                          |  <factor>
177
178 <factor>                ::= <element>
179                          |  <boolean>
180                          |  INTEGER_VALUE
181                          |  DOUBLE_VALUE
182                          |  CHAR_VALUE
183                          |  STRING_VALUE
184
185 <boolean>               ::= TRUE
186                          |  FALSE
187
188 <element>               ::= <ID_or_VectElem> POINT <ID_or_VectElem>
189                          |  THIS POINT <ID_or_VectElem>
190                          |  <ID_or_VectElem>
191                          |  <ID_or_VectElem> POINT <VectElem>
192                          |  <VectElem> POINT <ID_or_VectElem>
193                          |  <VectElem> POINT <VectElem>
194                          |  THIS POINT <VectElem>
195
```

```
196 <ID_or_VectElem>         ::= <id> LB INTEGER_VALUE RB
197                           |  <id>
198
199 <VectElem>                ::= <id> LB <vector_operation> RB
200                           |  <id> LB <id> RB
201
202 <id>                      ::= ID
203
204 <vector_operation>        ::= <id> <operation> INTEGER_VALUE
205                           |  INTEGER_VALUE <operation> <id>
206
207 <operation>               ::= PLUS
208                           |  MINUS
209
210 <main_block>              ::= MAIN <main_attributions> END_MAIN
211                           |  MAIN END_MAIN
212
213 <main_attributions>       ::= <main_attribution>
214                           |  <main_attribution> <main_attributions>
215
216 <main_attribution>        ::= <element> ASSIGN <factor>
```

Fonte: Autoria Própria